UiO : Department of Informatics
University of Oslo

Troubleshooting of Software Defined Networks

Automating Network Troubleshooting with SDN

Haris Sistek

Master's Thesis Spring 2015

Troubleshooting of Software Defined Networks

Haris Sistek

June 8, 2015

Abstract

Network troubleshooting is a field where automation is sorely needed. While the network has grown in many other ways since 1960s, the tools we use to troubleshoot and manage it have stayed very much the same. Could we use the programmability of SDN to automate this problem?

In this thesis work, we developed a prototype that would systematically troubleshoot the network with automation. The prototype automatically captures network behaviour, matches it against a network policy and creates its own test packets to troubleshoot the network policy, on a live system. When the prototype discovers a policy violation it will troubleshoot further down the SDN layers and move down a decision tree based on what the search finds on each layer.

The result showed that by the end of a troubleshooting search, the user would know what violating packet, packet path, policy rule, devices, flow entries and ports were the cause of the network problem. The results also showed that the automation of network troubleshooting could cause unexpected behaviour, and that some results were closely tied to timing between different POX controller applications.

By using the prototype, network operators could observe network wide traffic, both manually and automatically test that certain policy conditions were uphold on their network. The developed prototype, **sdn_dump**, helps solve network problems with automation by pinpointing violating behaviour and the network path it takes. Expanding the prototype to include troubleshooting of more SDN layers, and expanding the scope of the descriptive language should be a priority as future work of this thesis.

Acknowledgements

Firstly I want to thank my supervisor Desta Haileselassie Hagos for being patient, helpful and understanding in a stressful period. I will always be grateful for his dedication in helping me write and finish this thesis.

I want to thank the University of Oslo and Oslo and Akershus University College for providing resources, facilities and great times trough out the years I have been a bachelor and master student. I want to thank all my teachers, lectures and fellow students trough out the years, with their help I have grown both as a student and person. And I want to thank my fellow students in the NSA program for sharing 2 great years.

Lastly I want to thank my father, mother and little sister. They have been my bedrock this last year. Their love and encouragement helped me immensely.

Thank you all,
Haris

Contents

List of Figures

Listings

Chapter 1

Introduction

Networks and the complexity of the Internet have been expanding since their creation in the 1960s. While the network has grown in many other ways, the tools we use to troubleshoot and manage it have stayed very much the same (e.g. ping, traceroute, etc.).[1] Today's network operators/admins solve most of their network problems with what we can call manual troubleshooting. This involves the use of multiple tools in order to triangulate the problem area in the network. Manually troubleshooting a network problem takes time, while solving it can be comparably easy. A report done by H. Zeng and others, asked experience network operators about daily occupation activities. Some of these questions regarded troubleshooting. 24.6% admins answered that tickets take over an hour to solve[2, 1]. The most common problems were caused by hardware and software failures, and they were usually manifesting as reachability and latency problems[1]. Furthermore, the survey showed that when asking network operators what tool or advancement they most wish for, they answered "automated troubleshooting"[2, 1]. Solving such a problem will have a high return of investment (ROI) for the company and the operators that work for it. It could make the human cost of operating a network more efficient, and simplify how time and energy is spent. Ultimately, let operators use more time on solving the problem, rather than extensive time searching for it.

Brandon Hellers paper[2] shows how Software Defined Network (SDN) layering can be used to solve automatic troubleshooting problem. The paper shows how SDN is divided into multiple state layers that each represent a certain segment or view of the network (policy view, logical view, etc.) and how to use it to our advantage. Using this knowledge, they explain an optimal view of step by step troubleshooting a network. Each state layer will have equivalence checks that decides which way to go further, e.g. *"A: Does the network Behaviour Match the Policy?"*[2]. The fascinating thing with this approach is that we are left with a binary tree of decisions we can follow all the way till we identify the network problem.

1.1 Motivation

Brandon Heller and his fellow associates[2] theorize that using SDN and its layered approach to networking can be used to automate and systematically troubleshoot a network. This thesis will show how implementing their troubleshooting logic will handle common network problems, and the report will try to answer some of Heller's unanswered questions[2], including some of my own.

1.2 Problem Statement

This report focuses on leveraging the layered structure and programmability of SDN networks[2] for developing a troubleshooting tool. The report will demonstrate how this solution will reduce the number of actions a network operator must take before finding the source of a network problem, while also measuring such a solution's overarching performance. The thesis attempts at developing a troubleshooting tool that implements the troubleshooting logic based on Heller[2] previous work. This logic is implemented by incorporating already known debugging tools and concepts such as NDB[3], ATPG[1] and match them against a descriptive policy language developed during this thesis.

Keeping the above in mind, then the following are the problem statements of this thesis work:

How can we automatically troubleshoot networks using the layered structure of SDN?

This report tries to address the following research questions to further reach an answer to the main problem statement:

- *How can we automate troubleshooting using SDN Layering?*
- *How can we minimize or pinpoint the trouble/search area of a network problem using automation?*
- *Analyse the qualities of such a solution*
  - *Gauge the correctness of the solution output*
  - *Measure the fail-over rate of the tool*

Chapter 2

Background

This chapter will introduce specific literature, tools and concepts related to solving the above mentioned thesis problem as well as the tool implemented.

2.1 Software Defined Networks

2.1.1 Internet History

The tremendous growth of networking has contributed to constant expansion of Internet applications and services. We can instantly share video, audio, messaging and much more. The growth at the top layer of the OSI model has been tremendous. To keep up with this progress, our hardware has improved (Moore's law). We develop more powerful servers, routers, cables and switches. While this growth is great and the internet wouldn't be the same without it, the problem that remains is that the network, the fundamental thing keeping it all alive has stagnated. We still use the same internet protocols and technologies developed decades ago: TCP (1974), UDP (1980), DNS (1983), OSPF (1998). The main issue is that implementing, coming to a consensus and distributing a new network technology is multitudes more complicated than installing a new application or hardware to your site.

2.1.2 How Networks Work

In order to comprehend how networks function, below I will provide definitions of terminology used in this thesis work.

Hosts

A host is a network device that the user can access. Hosts are assigned an IP address which is used to communicate with other hosts on the network. We can distinguish between hosts and virtual hosts where the former is a hardware device and the latter a virtual machine.

Packet

Packet is structured data sent and received over the network. The data structure is divided into header, payload and trailer[4]. The header shows vital information such as destination, source, protocol, length and more depending on the protocol[4]. The payload is the actual data sent over the network with this packet[4]. The trailer is used to signal an end of the packet[4].

Switch

The switch device forwards packets between a LAN[5] (grouped hosts) and trunks multiple LAN together. The switch works on the second layer of the network (data layer)[5], while more modern switches, such as L3 learning switch incorporate the third layer (link layer) and use multiple protocols[5].

Router

The router is a device located between different networks[5]. It uses the packet header information and its routing table to decide the next hop the packet should take[5]. The routers work as a gateway between networks and calculate the best route a host should take to reach other hosts[5].

Network Policy

A network policy is a set of conditions, settings and constraints which can be viewed as policy rules that tells us when a network connection is allowed or not on the specified network[6]. The conditions (e.g. time condition) can be viewed as a property that must be true or false for the policy rule to apply, and the constraints can be additional variations (e.g. only apply for TCP connections) to each condition[6]. The combination of the two will create a policy rule (e.g. Time condition that only applies to TCP connections).

2.1.3 Traditional Networks

The problem with traditional networks lies at a few key places. Today's network hardware is usually what we call closed equipments with heterogeneous vendor specific devices. The software comes bundled in with the hardware, and the hardware uses vendor-specific interfaces[7, 8, 9]. The next issue is standardization. For the internet to work, we need to have standardized protocols, so that we know what behaviour we can expect from the other entities in the network. This is why we have organizations like ISO and IEC whose task is come to an agreement and decides what the standard should be. However such a process ensures a slow protocol standardization in the network. This stagnates network innovation. In practice, hardware creators write equipment code, and protocols have long vetting periods before they can be implemented across the network. Routers and routing protocols are not only hard to design and develop, but also need

to be verify for correctness[8]. This is what we call a vertical development approach and it has an impact across the network[10, 11]. The vertical development approach leaves us with very few people who innovate at the network level.

Traditional networks are also hard to maintain. We need network operators to manage the network and this is very expensive. This human element in maintaining the network also brings with it what we call human errors which can create network downtimes. Software and hardware bring with them a certain number of bugs and vulnerabilities which transfers over the different layers of the network. This can make them so much harder to debug than just traditional software bugs.

To summarize, slow network side innovation is identified by, but not limited to:

- Closed Hardware and Equipment.
- Long Standardization Process.
- Difficult Management/Maintaining Process.
- Bugs concatenate and transfer over multiple layers.

2.1.4 SDN History

SDN is part of a long history of trying to make the network more open to innovation by making it more programmable and friendly to network operators. In order to achieve this, the main goal has been to separate the control plane and data plane[7].

Main Goals

SDN has developed over a timespan of more than 20 years[7]. It started with the vision of a programmable and centralized network and has kept evolving. SDN has throughout its history had two main goals and features in mind. Firstly to separate the control and data plane, and centralize the control plane[7]. This way the control plane will handle routing network traffic, and the data plane will forward traffic based the control plane settings, restraints and conditions. Secondly, the goal has been to centralize the control plane that will then control multiple distributed data plane devices[7] such as switches and routers.

2.1.5 History and Time line of SDN

The young time line of SDN:

- Early 1990s - Started with Active Network.
- Early 2000s - Working on separating Control and Data plane.
- 2007 - OpenFlow API and Network OS were developed.

Active Network

The Active Network in the 90s was where we first saw the insurgence of ***programmable interfaces and network API***[7].

Separation of Control and Data Plane

In the early 2000s we incorporated lessons learned from the 90s. At the same time and in increased network traffic, and search for higher network performance and reliability spurred the development of a better networking solution[7].

The network landscape of the new millennia initiated the search for that solution. Researchers and network operators saw the increased use of devices that combined the two planes, but at the same time increased the complexity of managing the network[7]. They started by looking for a way to separate the control and data plane. This led to developing progress in three fields; ***an open interface between the two planes, centralized network control and distributed state management***[7].

OpenFlow API

In 2007 the OpenFlow API and network OS were developed[7]. OpenFlow (OF) was immediately deployable and launched SDN development forward. Later, controller platforms like NOX made it easy to expand and create new controller applications, while using familiar high level languages like C++ and Python. NOX has since diverged into NOX (c++) and POX (python), while other like Ryu, Floodlight and Beacon have come forward.

The significant difference with OpenFlow is that a switch using OpenFlow tables with packet handling rules come with pattern matches and appropriate actions[7]. OpenFlow actions are drop, flood, forward to interface, modify header, packet/traffic tracking counters and rule priorities[7]. When a OpenFlow switch reviews a packet, it will match it against a pattern, decide which rule to apply based on match and priority (highest first) and then perform appropriate action, while of course increasing the tracking counters. With OpenFlow the network operators and researchers had a solution for separation of control and data plane, by centralizing the control plane while keeping the data plane distributed.

2.1.6 History: What is SDN?

Today's traditional network has two functions. The data plane is responsible mainly for forwards traffic, while the control plane mainly routes traffic. Often these two function are integrated into the same hardware device (**Figure** 2.1), which increases the configuration complexity for a network operator.

In order to route traffic, the control plane needs to compute the routes and

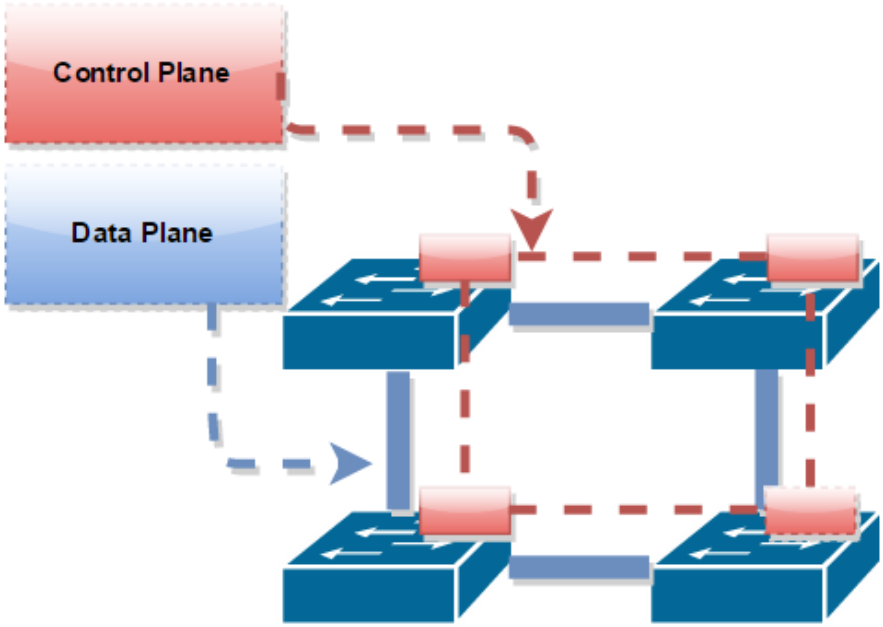

Figure 2.1: Traditional Network

update routing tables. These two functions are done in distributed routers. We often also talk about a management plane, this is done by a network operator. Here we specify what network policy and behaviour we want.

On a traditional network this may be very tedious and slow work where the operator has to interface with each machine individually and decide its behaviour. However, on a SDN network the control plane is centralized into one controller[8, 11, 7], which has system wide control to each network node (**Figure** 2.2). This feature gives the network operator network wide control over how to build and configure the network[8, 11, 7]. It also centralizes and streamlines the management plane process we had to do on the traditional network. This takes us from a vertically integrated network, with slow innovation to a horizontal/open-interface network with relative faster innovation and better control[10, 11].

2.1.7 History: How SDN works

The OpenFlow protocol gives us a way to simply handle packets using events and rules[12, 13]. We can match a packet using patterns, then give the switch an action (forward, drop) for that match[12, 13]. OpenFlow also gives us the ability to prioritize different matches and monitor the traffic using different counters provided by the protocol(number of packets)[12, 13].

The programmable controller gives us an unique ability. The controller is built upon a network OS that receives events from switches (packet in,

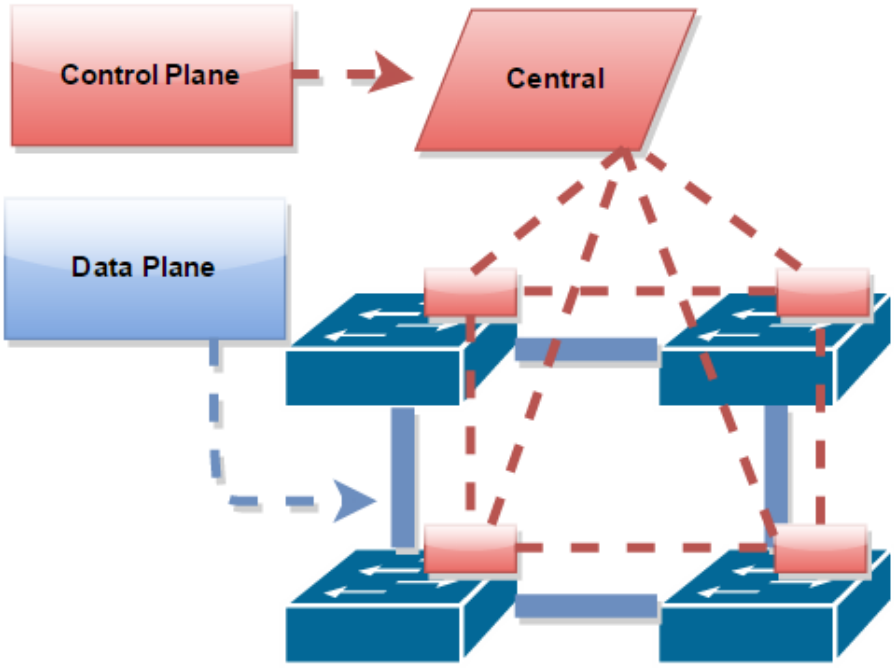

Figure 2.2: Central Control Plane

device up) and can give new commands to switches (install new rule, send packet)[12, 13]. We distinguish between reactive and proactive SDN[14]. A reactive SDN is closer to the experience I have had using POX controller. The controller passively listens to events. When an event is triggered an action takes place in your code. The proactive SDN programs and controls switches using global knowledge of the network through discoveries and updates from the switches. An example could be to program the switches before the traffic comes and anticipate an action for each traffic match possible that could take place on that switch. The proactive solution looks and performs more like the routing tables of today's networks[14], with higher performance on high frequency networks, while the reactive solution is more flexible in how it can handle traffic. That would mean that if no rule in the forwarding table can apply, the reactive solution would be to create a rule for that packet, which of course means more processing time[14]. Perhaps a hybrid[14] solution may combine the flexibility of reactive and performance of the proactive one.

2.1.8 Horizontal Development Approach

The key ingredient as to why SDN is such an exciting new technology, is the move from a vertical developed network onto a horizontal one.

2.1.9 Exciting SDN Technologies

The possibility of an SDN platform that is running network applications are numerous. We have seen applications that improve bandwidth management to a network utilization of 90-95%[8]. Applications that produce easier tenanted networks[8], by simplyfing the interaction between physical VLANs and virtual networks. Applications that reduce latency on gaming applications while the device (this case a laptop) is mobile and switching multiple subnets[8].

2.1.10 Advantages and Challenges with SDN

The advantages that follow from this paradigm shift is that now we can program the control plane in high level languages like C++, Java and Python. This means a more **programmable network**.

Moore's Law - We could move the computing power (CPU, RAM) from the routers onto the centralized controller. Having a powerful controller that routes the traffic could give us a Route Compiler In The Sky(RCITS)[8], and leaves us with more cost effective hardware that focus more on just forwarding traffic.

Global optimization - Using the controller and the programmable network we could possibly one day have a global optimization of routing[8], with one central controller for the entire network.

Hardware as a Commodity - Taking the software out of the hardware leaves us with less expensive and more comparable hardware[8], where we compare on CPU, RAM and not on the software the vendor includes with the hardware. The ideal hardware should be simple, vendor-neutral and future proof[9]. This shift takes network hardware closer to the PC market where we look at hardware and software as more separate entities[10, 11].

SDN still holds a few key challenges:

- Scalability - A network can keep hundred to thousand switches.
- Consistency - Ensuring the same view for the multiple different network operators and controllers over the same network topology.
- Security - A compromised controller is a danger to the whole flow of the network.
- Performance - A controller can't outperform a switch in speed and it uses time to process packets coming in to the controller, which leads to delays.
- Software bugs - Controller application are programmable and this eventually leads to software bugs.

2.1.11 MininNet

MiniNet is an OpenSource project that creates a SDN using OpenFlow[15]. It is easy to use, understand and program a wanted network topology. The thesis used the provided Mininet VM from their web page during this project[16], where we tested and developed the troubleshooting application.

2.1.12 SDN Controllers

The controller is controlling the entire network. It uses OpenFlow protocol[13] to communicate with the network switches and routers. The controller will have different applications that perform network task. These applications can be combined, it is this that makes the SDN look so flexible and programmable. For instance we can run an application that makes the network switches behave as *l2* or *l3 learning* switches, network logger, network load balancer, and much more. But first you have to choose a controller. As mentioned before there are many variants now such as NOX, POX, Ryu, Floodlight and Beacon. In this thesis POX was chosen.

2.1.13 POX

POX is a controller platform that is used to develop controller applications using the OpenFlow protocol and API[12, 17]. POX is now the python equivalent of NOX[17, 18]. The legacy version of NOX used to support both languages[19], but they have since diverged into the two different controller platforms we have today.

2.1.14 POX and Python

In this thesis work Python and POX were chosen as the language for the implementation. POX uses Python, NOX uses C++, so the decision was easy given my relative knowledge of Python vs. C++. Python is often regarded as the simpler language to both write and read, for non and programmers alike. Python has a rich library of online documentation and helpful guides, while POX is less documented and it was harder to find helpful resources. In the end Python and POX were chosen as to not allocate more time in learning a whole new language.

2.2 Troubleshooting Networks

2.2.1 Traditional Troubleshooting

Traditional network troubleshooting is usually done through use of various different software that cover different areas of the network. These tools are often as simple as *ping, traceroute, tcpdump, netstat* and many more[1]. The network operator needs experience with each of these different tools so as to learn how they work and interact. An operator will usually try

to triangulate the error by issuing specific commands with two different tools which pinpoint the error. This part of troubleshooting is the most tedious and time consuming process. While a network fix could be to change a line, on a given device, in a given configuration file, finding that line could take hours [1, 2]. Seeing as how the human cost is one of the biggest expenses in managing a traditional network, a great cost saving opportunity emerges as well as making the troubleshooting process easier for the network operators. When asking experienced network operators what tool they most wanted to be developed, they answered "automatic troubleshooting"[1, 2]. Going from using multiple different tools and spending extensive periods of time in finding the problem, to just start a automatic debug tool searching for you, allows the operators to fix problems rather than searching for them.

2.3 SDN State Layers

SDN segmentations give us the advantage of having a layered representation of the network (**Figure** 2.3). Each layer represents and controls its own domain over the network. The interesting part is that "bugs" created in one layer will manifest/trigger an error between itself and another layer[2]. E.g. A "bug" in physical view would show erroneous behaviour in the connection between 'logical view <-> physical view' and/or 'physical view <-> device state'.

2.3.1 Policy Layer

A network policy outlines certain conditions and settings which describe how we want the network to behave. It describes network access and authorization, when one can connect or disconnect from it and much more. The errors we find here are more often than not human errors. This means that the errors are related to configuration and parsing[2].

2.3.2 Logical View

This is the abstract topology as we, the users, see it. The logical and physical topology do not have to be the same. For instance two virtual switches can actually represent one physical switch, or one virtual switch can represent hundreds of physical switches. The possibilities here are both numerous and complicated at the same time, especially when a network has multiple controllers. The question we need to answer here is: "How do we ensure the same logical view to both controllers at all times?" Errors found here can be policy mistranslation[2].

2.3.3 Physical View

This is the physical topology of the network. It consists of switches, routers and connections between them. This should be the accurate view

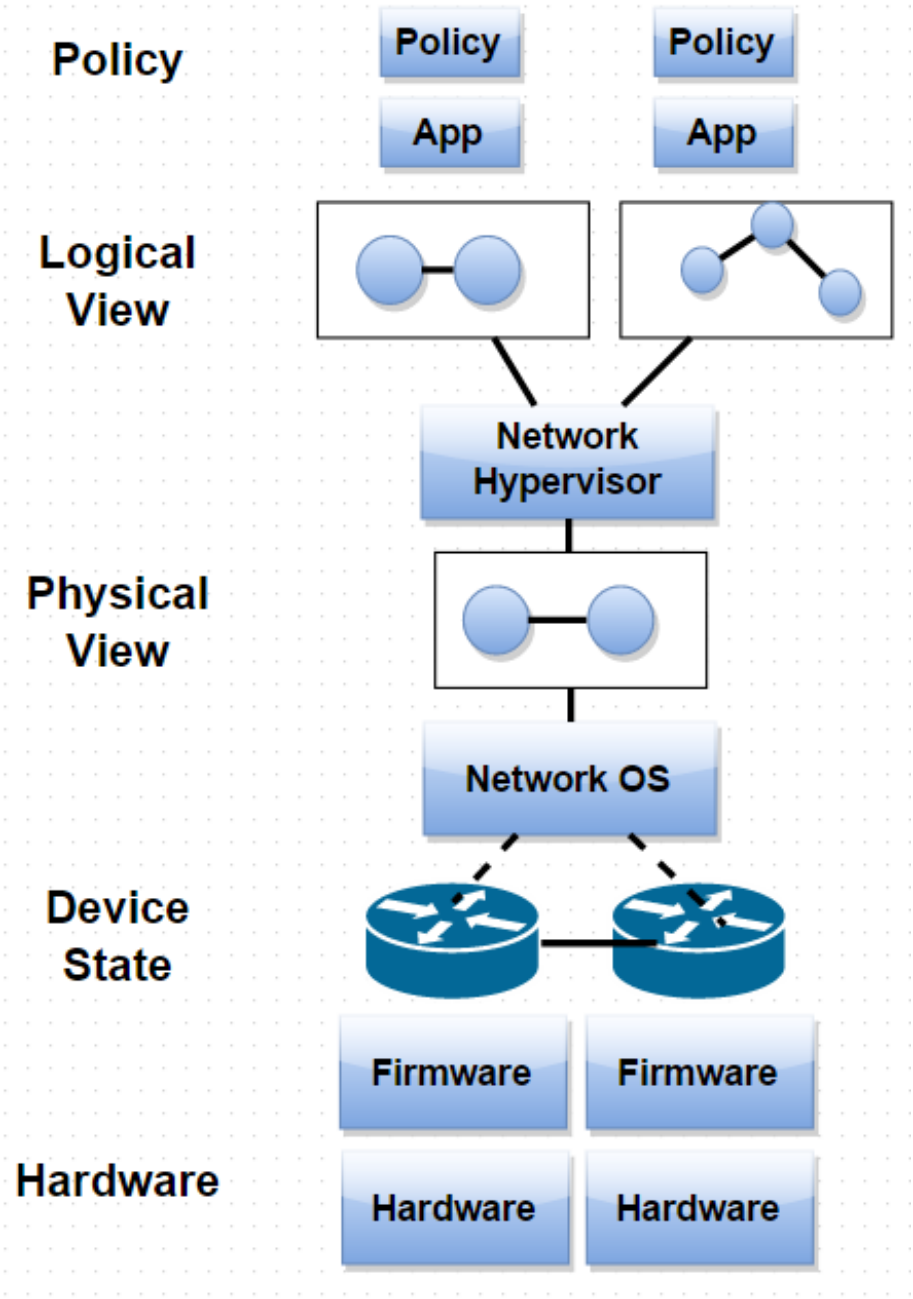

Figure 2.3: SDN Layers [2]

of the physical network. Errors found here can be fail-over logic and synchronization bugs[2].

2.3.4 Device state

This is the firmware on our devices. This is usually where forwarding tables are mapped out. Errors found here are often low level issues. Things like memory corruption on hardware and register misconfiguration[2]

2.4 State Equivalence Check and Decision Tree

Using state layer knowledge of SDN allows us to systematically perform step by step troubleshooting of a network, through a number of equivalence checks that in the end create a binary tree of decisions (**Figure** 2.4).

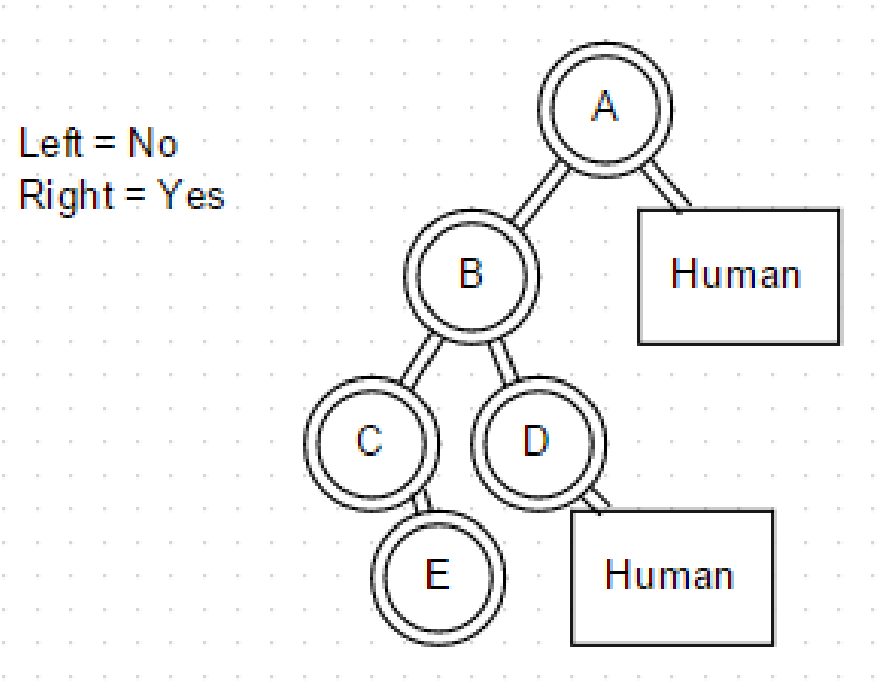

Figure 2.4: Decision Tree of Questions

Each equivalence check, checks between two state layers. If the layers match, then the problem is outside the scope of the two layers. However if they don't match we search a subset of the two layers[2].

2.4.1 Question A

"*A: Does the Actual Network Behaviour Match the Policy?*" If *yes*, then the policy itself is the problem, and a human must resolve the discrepancy. If *no*, we continue." [2] We need to check actual packets passing through the network against the network policy that the operators have set. By the end this question, we should know what policy rule is being violated and from what network behaviour this violation stems from. **Suggested existing tools:** ATPG and NDB can partially solve this check, but cannot check directly against a high level network policy[2].

Relation to thesis

This will be the starting point of the thesis design and implementation. By distinguishing if actual behaviour and intended behaviour match, we can

automatically move down the next step in the decision tree. We should also inform the user what policy rule is being violated, and what network behaviour is causing the violation (devices, packets and packet paths).

2.4.2 Question B

"*B: Does the Device State Match the Policy?*" If *yes*, then the control plane is behaving properly; the problem must be lower in the stack, either in firmware code, or a hardware component itself (e.g., a link, port, or table)"[2] Here we are searching for common network problems such as forwarding loops, black holes and disconnectivity issues[2]. If the answer is *yes*, go to question D. If *no*, then go to question C. **Suggested existing tools:** Anteater, Header Space Analysis and VeriFlow[2].

Relation to thesis

Once our program finds a discrepancy in the previous question, the automation will start searching between device state and policy.

2.4.3 Question C

"*C: Does the Physical View Match the Device State?*" If *yes*, the actual device state is correctly synchronized with the physical view, then the problem must be above"[2] If yes go to **question E**. If *not*, then the problem lies between Physical View and Device State (Network OS). This is the least researched question of them all, and will require new tools for a proper solution[2]. **Suggested existing tools:** OFRewind[2]

2.4.4 Question D

"*D: Does the Device State Match the Hardware?*"[2] If *yes*, then hardware is to blame, and we will need a human to diagnose and resolve the problem. If *no*, then its a firmware problem. **Suggested existing tools:** SOFT[2].

Relation to thesis

If the troubleshooting process ends here, it has reached an end point, the leaf of the decision tree. We want to show the user how the high level policy has been implemented across the network on the different network devices. This entails showing the user low level configuration of the problem causing devices (devices that violating packets have travelled trough), that would mean showing policy violating devices, flow entries and port sets(inport and outport) on those network devices.

2.4.5 Question E

"*E: Does the Logical View Match the Physical View?*"[2] If *yes*, then the applications running are to blame. If *no*, then the problem is between the

logical and physical view (network hypervisor) **Suggested existing tools:** Some way to do a correspondence check logical vs. physical topology[2].

2.5 Troubleshooting Automation

To automate troubleshooting of a network we need 3 things. a) The actual behaviour of the network, b) the ability to simulate creation of behaviour and c) we need to know the operators intent with the network[2], i.e. the intended behaviour of the network.

2.5.1 Actual Behaviour

We can define this as the observable network and how it behaves. How each packets get sent, forwarded, dropped and received. If we send a ping from *Host1* to *Host2*, and we observe that it jumps from *Switch3* and *Switch4*, and back again we have observed a little piece of the networks actual behaviour. All network communications combined provides us with bigger sample of the actual behaviour. With tools such as NDB[3] we can observe the actual behaviour of the network as described above, and have the ability to see the step by step path the packet takes. With actual network behaviour we have reference point to use when we are troubleshooting the network.

2.5.2 Intended Behaviour

This is how a network operator intends for the network to behave. It can also be called the network policy. It should be a clear description of how the networks should behave and deviation from this would usually be unwanted behaviour. In the traditional network, the network policy would be a combination of low level configs across the different network devices. For instance drop all packet from and to Facebook could be such a configuration and therefore a part of the policy. However, with the shift to SDN we have the ability to centralize the network policy in one place. "In an SDN, policy can be a fist-class citizen; intent can be *explicitly* expressed at the policy layer, while the code compiles the policy description in lower-level configuration."[2] With the actual and intended behaviour we have two reference points of the network. Using the two we can compare one against the other, and see if the actual behaviour violates the intended one.

2.5.3 Automating Behaviour

Automating or rather simulating behaviour for troubleshooting is complicated. We need some form of description to reference and then create behaviour from. We can use the policy description itself as the reference to automate intended behaviour by translating policy rules into network traffic using automation, however this brings with it some problems. What if the policy is the problem, meaning what if the described intention of the

operator isn't really what he intended for, e.g. the policy itself is the cause of network problems.

2.6 Troubleshooting Tools and Concepts

2.6.1 ATPG: Automatic Test Packet Generation

An ATPG reads router specifications and generates tests packets to test each link and table rule of the network[1]. This way it can triangulate the bug into a single node, link or forwarding table entry at the network[2]. ATPG excels at finding both functional(forwarding rules, link rules, drop rules) and performance(congestions, bandwidth, reachability) based problems on the network[1]. ATPG can verify reachability and performance by testing nodes through sending of test packets[1, 2]. It does this by learning the full list of the forwarding tables, creating a model of this and producing a set of test packets that will test all of the forwarding rules[2, 1]. It locates and pinpoints problem areas by looking for packet drops that can indicate software and hardware errors[1]. This tool can be used to test a network's low-level configurations and see if problems can be created. Using this tool we could partially solve the first equivalence check of the binary tree Heller proposed[2]. However it does not have the ability to check against a high-level policy as Heller noted[2]. On the other hand, ATPG or a similar tool could solve the issue of automating network behaviour that we can use for testing, thereby lessening the burden on operators further.

2.6.2 NDB: The Network Debugger

NDB is meant to debug a network in much the same way as how gdb[20] works. It monitors your network, similarly to a network wide tcpdump[2, 3], but implements two interesting features: you can add breakpoints and backtrace your packets on the network[3]. This way it can pinpoint the sequence of events that lead to network errors, much in the same way you would use gdb in for instance C-programming. Using NDB or a similar tool would enable automatic capturing the actual network behaviour, which we could later match against a network policy, thereby partially solving the automation of question A: "*Does the Actual Network Behaviour Match the Policy?*"[2].

NDB: Breakpoints

NDB followed with the gdb fashion of breakpoints. They implemented control actions such as breakpoints, watch, backtrace, single step and continue[3] The result from this is that the user can view a packets path to the network error that it is creating.

2.6.3 Tcpdump

Tcpdump is a monitoring tool at end-hosts. It captures packets as they enter and leave the hosts, and prints the packet content with a timestamp at the time of capture[21]. Tcpdump is often used by network operator because of the utility it provides. The operator can assign and combine boolean expressions (tcpdump src 10.0.0.1 and dst 10.0.0.2) so to filter specify packet properties that you are interested in. Having a network wide tcpdump, such as NDB[2, 3] is a way of capturing actual network behaviour.

2.6.4 Ping

Ping is one of the most used tools in network administration[1]. A ping packet uses the ICMP protocol[22]. It works by sending a ECHO_REQUEST to a destination and waiting for a reply such as ECHO_REPLY or error messages such as TIME_EXCEEDED, DESTINATION_UNCREACHABLE and more. This is the simplest way for an operator to check if it can reach a certain destination from his ping location.

2.6.5 Traceroute

Tracroute works similar to a ping and is also one of the most used tools by network operators for troubleshooting[1]. You give it a destination and it sends packets to that destination. The big addition to tracroute is that it uses a small TTL[23] to know and show the operator the packet path that is being taken from this source to that destination host.

2.6.6 OFRewind

OFRewind set out to supply a tool that network operators were severely lacking. While software developers have good debuggers and varied tools, the network operators are left to use crude and limited tools[1, 24]. With the help of SDN and the OpenFlow protocol, the developers of OFRewind set out to give operators the ability to have: "scalable, multi-granularity, temporally consistent recording and coordinated replay in a network, with finegrained, dynamic, centrally orchestrated control over both record and replay."[24] This way operators can reproduce errors and the packet paths that created them.

The Main Advantage

The main advantage OFRewind has over similar recoding tools (Netflow, tcpdump) is that it can cast a wider net[24]. Tcpdump and Netflow are usually recording on a single interface or switch port. OFRewind also has the advantage that you can follow the packet path to see the starting point of the problem, while not significantly affecting the scalability of OpenFlow in a negative way[24]. Heller states that of all the equivalence questions, C "*Does the Physical View Match the Device State?*" is the least researched[2]. However he states that OFRewind provides us a start on answering it[2].

OFRewind similareties with NDB

OFRewind is in many ways similar to the NDB[3] tool, and is internally using it in collecting traffic from the dataplane[24]. OFRewind does have a few limitations. It does not support flow and packet sampling, partial packets forwarding in traffic, flow cut-offs when reaching a certain traffic amount and safeguards against replay inaccuracies[24].

2.6.7 Anteater

Anteater leaves diagnosing the control plane for other tools and focuses on diagnosing problems through static analysis of the data plane[25]. By focusing on the data plane it can find bugs that we cannot directly view from the configuration files[25]. By looking on the data plane state of network devices it can analyse close to the actual network behaviour[25]. Its main aim is to find broad network problems by analysing forwarding tables across the networks devices, and then looking for problems by checking the tables against invariants such as[25]:

- Loop-free forwarding
- Connectivity/Reachability
- Consistency in network behaviour

Using Anteater we can solve equivalence question B "*Does the Device State Match the Policy?*" since it checks connectivity against the data plane states[2] and find other device state issues such as black holes and forwarding loops.

2.6.8 Header Space Analysis

Header Space Analysis is a general framework for analysing packet headers[26]. The main goal is to reduce the complexity of troubleshooting common network problems by providing network operators an easier way to analyse their live production networks[26]. It is done by giving the operator answers regardless of the network protocol in use on the network, answers on problems such as host/tenant isolation, reachability, loop detection and traffic leakage[26]. By using Header Space Analysis or its developed tool Hassel[26], we could solve question B "*Does the Device State Match the Policy?*"[2] since it will find errors such as loop detection and reachability. Then it checks for correctness problems at data plane through a few steps. Firstly it models packets in the form of header space[27]. Then it models network devices as functions that change header spaces, for example a router will in addition to forwarding packets, reduce the TTL in the packet header[27]. Finally, through a set of header space algebra we can compute header spaces that result in intersections, complementation, difference and checking for subsets and network equality conditions[27]. So through tools such as Header Space Analysis we can discover problems in the data plane state. Nick Feamster[27] notes that the tool only gets the

current snapshot of the data plane, and will not tell you what may happen if the control plane changes the data plane state after such a snapshot.

Chapter 3

Approach

This chapter will provide a step-by-step approach, showcasing the actions taken in order to answer the given problem statement in the introduction chapter. We will go through three different phases in this thesis: design, implementation and experimentation. This chapter aims to give an overview of each one.

3.1 Approach Waypoints

The thesis has a number of waypoints and will go through three phases to give a clear roadmap of how the design, implementation and experiments phases were conducted.

1. Design prototype:
    (a) Question A
        i. Identify the three different behaviours needed for troubleshooting automation
        ii. Design and model:
            A. Capturing intended behaviour
            B. How to match actual behaviour against intended behaviour
            C. Creating automatic testing behaviour
        iii. Show Question A model overview
    (b) Question B
        i. Look for disconnectivity issues
        ii. Search for common network reachability problems
    (c) Question D
        i. Show the user lower level configurations that are causing network problems
        ii. Show how they are related to the higher level policy violations by giving policy rule, network device and switch ports & flows

2. Implement prototype:
    (a) Question A
        i. Implement a solution for:
            A. Capturing actual behaviour
            B. Capturing intended behaviour
            C. Creating automatic testing behaviour
            D. Decision maker to move down the tree
    (b) Question B
        i. Check for disconnected network devices
        ii. Search for forwarding loops with spanning tree protocol
        iii. Decision maker to move down the tree
    (c) Question D
        i. Request flow tables from network devices
        ii. Compare and filter out irrelevant entries
3. Experiment with prototype:
    (a) Measure and Demonstrate System Behaviour
    (b) Analyse adequacies and strengths
    (c) Evaluate prototype

3.2 Design

The design is intended to show the concept and model of how the thesis intends to automate using the systematic layering of SDN. Heller has previously provided us with a set of equivalence checks which need to be answered for this to work[2]. This thesis will primarily focus on the two first checks given by Heller[2] Question A: "Does the Actual Network Behaviour Match the Policy?" and Question B: "Does the Device State Match the Policy?". This means that both the design and implementation will be two parted, and each part devoted to one or the other equivalence check.

3.2.1 Question A: Design

Question A: "Does the Actual Network Behaviour Match the Policy?"[2]. This part will focus on explaining the approach in automating the solution to question A. It will specify what the system does and and how it will perform the tasks assigned. Meaning how do we capture packets, check them against rules and even create packets that will test these rules? These questions will be covered in this section.

3.2.2 Question B: Design

Question B: "Does the device state match the policy?"[2]. How do we show the user if the device state matches the policy? How do we go about finding errors such as forwarding loops, black holes and disconnection issues. These questions will be answered in this section.

3.2.3 Question D: Design

Question D: "Does the Device State Match the Hardware?"[2]. The aim here is to show the operator the low level switch flow entries that are a cause of a higher level network policy violation, such as we find in Question A. This would entail showing the operator what devices, port sets (inport, outport) and flow entries are causing the network problem from the device standpoint.

3.3 Implementation

The implementation will show how the design was implemented in Python using POX[17] and tested on different Mininet[15] topologies. Though this section will list a few code examples, the full code source will be listed in appendix C of this thesis and the public repository will be available at: **https://github.com/HarisSistek/public_sdntroubleshoot**.

3.3.1 Question A: Implementation

The implementation of Question A will need 3 python classes. One that is the decision maker and network wide listener, the second one should be able to read a policy description and check it against the packets that the previous class is listening and capturing, and the third should create automatic packets from the policy description and test if they create policy violations or not. By the end of Question A, the operator should be informed about what policy rule is being violated and on what network devices these violations are occurring.

3.3.2 Question B: Implementation

Most of Question B will be handled by the network wide listener class deigned during Question A, since it is already closest to the problem at hand. We will also use a pre-made POX module/application that will execute a spanning tree protocol that removes loops on the network. After the spanning tree we will initiative a connection check on the switches to look for disconnectivity issues.

3.3.3 Question D: Implementation

Question D is also handled by the network wide listener. Once we reach this part of the decision tree the controller will request the flow statistics

from the network switches, and we will use that information to reduce the scope of the problem. By the end of Question D, the user should know what rule, packets, packet paths (Question A), what switch (Question A/D), and what switch ports and flow entries (Question D) are creating a network problem.

3.4 Experiments

The aim with the experiment is to test the features of the prototype. The experiment should be constructed as such that we can test each equivalence check (Question A, B and D) finding and not finding a violation, and what we extract from information the tool provides in these situations.

3.4.1 Experiment with the monitoring feature

These experiments should aim to test the prototypes ability to capture network traffic across the network. The experiments should aim to use both different packet types (packet protocols), and different network sizes (number of network devices).

3.4.2 Experiment with the violation finding feature

These experiments should aim to test that the prototype can capture violations to different policy rules. Firstly, this means that Question A should discover network behaviour that violates policy rules related to time, date and data sent/received. The experiments should also show the violations found on Question B and D, and not just the top layer.

3.4.3 Experiment with the automating feature

The experiments here should aim at testing the prototypes ability to create different kinds of packets to test the described policy rules.

3.4.4 Experiment with tools that alter network behaviour

This experiment should show how the tools interacts with another POX controller applications. These applications should in essence be enforcing the policy we are describing with our policy language. From the experiments we expect to see violations occur when the applications are not running, but no policy violations to occur when they are.

Chapter 4

Design Results

This chapter will explain the design decisions taken during this thesis. It will show how we use SDN technologies to troubleshoot a network, and will serve as a road map for implementing the prototype. We start at the top of Hellers[2] binary decision tree(**Figure** 2.4) and work our way down.

4.1 Question A: Design

The first equivalence check at the top of the tree is *"A: Does the Actual Network Behaviour Match the Policy?"*[2] To answer this question we will need 3 things. The actual network behaviour, the intended behaviour, and to fully automate this check we would also need to automate/simulate network behaviour. Once we have the first two (actual and intended behaviour) we can match them against each other and look for policy violations. If we find one, notify the operator and program to move a step down the tree. If non are found, then we will need a human to diagnose the problem. With this design the operator will have the ability to just describe intended behaviour and let the program troubleshoot the network automatically from there. Ultimately we want the operator to start the **sdn_dump (Figures4.1 and 4.2)** module on the controller and let the network troubleshoot itself.

Once a violation is found the operator should be notified where it is, and **sdn_dump** should move one step down the decision tree until the location of the problem is pointed out. Each step in the tree should grow the a trace stack. In the trace, Question A will notify the user about what type of violation (e.g. Forwarding Violation) it is, what behaviour (devices, packets and packet paths), and policy rule was the violating cause. Question B would then troubleshoot its search area and add significant information to the stack and move us left or right down decision tree. Then if we are at Question D, which is an leaf of the tree (end node), Question D would add violating devices, device ports and flow entries to the stack. Using this trace stack, the operator can follow the trace down to the location of the error, from high level policy to low level configuration setting on the network device.

Listing 4.1: Example Violation Found

```
## Policy Violation Found: Violation-Type ##
> Packet: Packet Info
> Packet: Packet Info
  violates:
> Rule: Rule-text
... Next Stage ...
```

4.1.1 Question A - Part 1: Capture Actual Behaviour

The approach in capturing the actual behaviour is pretty simple. Have a passive listener at the controller and forward all network packets to the listener. It is a network wide **tcpdump** similar to what the NDB[2, 3] provides, but without the path awareness, packet backtracing and traffic breakpoints, my app will be called **sdn_dump**. Once the **sdn_dump** starts to receive packets it will print out relevant packet data for the operator to see, similarly to **tcpdump**. Then it will forward the packets to the the next module in the system (**violation_checker**)**(Figure 4.2)**. This module will check the packet against the intended network behaviour. The *sdn_dump* will be a separate module from the other two **(Figure 4.1)**, in order to capture packets and device stats independently from checking violations or automatic packet generation. By separating these functions we can give the tool multiple functionalities that the operator can use. The operator can command the tool to use a certain operation mode *(modes = 1 - listen for packets, 2 - listen and check packets, 3 - 2 + automate packets)*. This way the user should be able to leave out the generation of test packets by command, and have the ability to create his own network behaviour to test using the usual external tools such ping and iperf.

Listing 4.2: Example Listening Function

```
def _handlePacketIn(event):
        packet = event.parsed
        if packet == some_packet_type:
                handle_that_type
        elif packet == some_packet_type2:
                handle_that_type
        else:
                Unknown_packet_type
```

Since the packets that **sdn_dump** receives will enveloped differently depending on packet type we will need different methods to handle different type of packets with different methods and outputs. For instance, receiving an ICMP, ARP and UDP packet, would mean that three different, but similar outputs that we will later parse differently in the next section.

Listing 4.3: Example Outputs

```
<time> DevID: <devID> ICMP: pkt <hw-src> > <hw-dst>,
   ip <src> > <dst>: <icmp-type>
```

```
2 <time> DevID: <devID> ARP: pkt <hw-src> > <hw-dst>, hw
      <src> > <dst>: <arp-type>
3 <time> DevID: <devID> UDP: pkt <hw-src> > <hw-dst>, ip
      <src> > <dst>: srcp <sport> dstp <dport>
```

Once we have read a received packet we should send it for checking at the **violation_checker**.

Extracting device statistics

We should regularly request port statistics from the connected network devices. We can set it to be done after a certain number of packets have been sent and received on the network. As the network is sending and receiving packets, we should also map *IP-PORT-DEVICE* relationships such that we can later derive data used by each host. Once we have the relationship mapped and the port statistics for each device, we can send this information to the **violation_checker** to ensure that a data violation has not occurred.

Question A - Part 1: Summary

- Listen for new packets
- Parse the packets
- Print packet information
- Send packet information to **violations_checker**
- Map *IP-PORT-DEVICE* relations
- Extract port statistics per device
- Send port statistics to **violation_checker**
- Decide what to do based on response from **violation_checker**

4.1.2 Question A - Part 2: Describe Intended Behaviour

We now need a simple way for the operator to describe intended behaviour (policy), but that can later be used both for matching and generating automatic test packets. The approach would be to let the operator describe conditions and constraints at which point a host would be blocked on this controller controlled network. It is important to notice that we are simply talking about a descriptive language, it should describe the operators intent with the network and thereby also intended network behaviour. **This means that the language will never change how the actual network behaves other than create test packets from the description,** i.e. lower level configurations will stay intact. With this approach we can insure that actual network behaviour and the networks intended behaviour are completely independent and separated for the program.

The descriptive language should be close to human speech, but also make the syntax modular and reusable, and simple to parse later on. The policy document should start with the network operator assigning hosts to IP addresses that it can later use to refer to the policy rules, this part shouldn't be necessary, but should simplify the reading of policy rules.

H1 = 10.0.0.1
H2 = 10.0.0.2

Then each line starts with a major constraint, like Time, Date, Vlan, Data. The constraint comes with a description specific for that type of constraint:

Time 20:00 to 21:00
Date Fri to Sun

After the major constraint we need an additional description on what host or host-to-host specific connection we want to block.

Time 20:00 to 21:00 **block H1**
Time 20:00 to 21:00 **block H1 to 10.0.0.2**

We also need the ability to add constraints to rules by specifying protocol and port numbers that should be blocked on that condition.

Time 20:00 to 21:00 block H1 **prot UDP sport 1000 dport ***
Time 20:00 to 21:00 block H1 to H2 **prot TCP sport * dport 22**

Finally, provide the user the ability to make comment in the document using #-sign. The other major constraints should work in a similar fashion:

Date MON to FRI block H1 - similar function as time constraint

Some primitives won't have need for protocol and port specification:

Data 100 MB to H1 - H1 can't receive more than 100 MB
Data 100 KB from H2 - H2 can't send more than 100 KB
Vlan 10 has H1, H2, H3 - Vlan 10 has tree hosts H1, H2 and H3

Using the language we can capture the network operator intent and parse it into usable data, store it into python dictionaries and then later compare them against packets that are forwarded from **sdn_dump** to **violation_checker**. This way we can find packets that violates the intended behaviour.

Listing 4.4: Policy Example

```
1 # Hosts
2 h1 = 10.0.1.100
```

```
3  h2 = 10.0.1.101
4  h3 = 10.0.1.102
5
6  Time 10:00 to 23:00 block h1 to h2
7  Time 10:00 to 23:00 block 10.0.1.100 prot UDP
8  Time 10:00 to 23:00 block h2 prot TCP
9  Time 10:00 to 23:00 block h3 to 10.0.0.6 prot TCP
       sport 1000 dport 22
10 Time 10:00 to 23:00 block 10.0.0.6 to h3 prot TCP
       sport 22 dport 1000
```

Checking Time

Allows the user to check if the network blocks a connection or host during a certain time period. For instance ensure to block all connections to Facebook during work hours. We can check this by looking at the time the packet is received against the Time-rules time interval.

Checking Date

Similar to the Time primitive, check if the network blocks a connection or host during a day or interval of days. We can check this by looking at the day the packet is received against the Date-rules.

Checking Vlan

Checks that the tenant relationship for this VLAN is intact. We can check this by looking for behaviour that would violate this VLAN. For instance check that packets request and replies do not jump from one VLAN to another.

Checking Data

Check that a host or connection does not transmit or receive over a certain data amount threshold. Openflow keeps many counter including packets and bytes sent/received for each switch port and flow table[13]. Using this we can match the amount against the limit set by the operator in the description policy.

Question A - Part 2: Summary

- Describe conditions and constraints in the form of policy rules
- Parse and store the policy rules to be used for checking and automating packets

4.1.3 Question A - Part 3: Match Actual against Intended Behaviour

Once the **sdn_dump (Figure 4.2)** is running, it will listen for incoming packets. Once one is received, it is then forwarded to **violation_checker**. The packet should then be parsed into usable data, that we can match against rules in the policy files. We want to check as few rules as possible to minimise the time for finding a violation since the number of packets can become great. This can be done by disqualifying rules based on protocol type (TCP, ICMP, UDP). The next step would be to look at source and destination IP address and port numbers. Then finally to look at the major primitive (Time, Date) and see if they are violated.

Listing 4.5: Matching Example

```
1
2  def check_if_legal(packet_string): # string from
       sdn_dump
3    if proto_type in packet_string:
4          packet_data = extract_packet_data(
               packet_string)
5          bool_violation = check_if_proto_type(
               packet_data)
6          if bool_violation:
7         stored_packets.append(packet_data)
8         if check_for_violation(packet_data):
9           return True
10   elif proto_type2 in packet_string:
11         ....
12   else:
13         ....
```

When a packet has been checked by rules and found to violate the policy, we remember that packet by storing it in a list. If the next packet **violation_checker** received also violates one of the rules, we need to examine if that packet could either be a reply or a forwarding of that packet to another switch. This is important, because if a user is running a controller application that would drop a certain packet the application will have to see the packet be received to the switch from a host, but the packet should not be forwarded to or be replied to by another host. In practice, we would only see one instance of that packet on the network for the user application to work correctly. Any more instances of that packet on the network would create violations between actual and intended behaviour.

Matching port statistics against policy

The **violation_checker** can also receive mapped port data from **sdn_dump**. The function **check_if_ports_legal()** will be used to check the mapped *IP-PORT-DEVICE* data against the Data policy rules and see if any has been violated.

Violation is found

If a violation is finally found, notify the user which packet and host violates which rule, and return a boolean value to **sdn_dump** so that we can move down the decision tree. This means that even if there are more violations to be found, print out the stack trace of this packet violation rather than all the possible rule violations at once. This way the operator will need to solve one problem at a time.

Question A - Part 3: Summary

- Parse packets received from **sdn_dump** into useful data
- Match an already stored policy rules against the parsed packet
- Remember packet that can create a violation in the future
- Check port data against Data rules
- Return boolean value to **sdn_dump** if two packets or port data have created a violation

4.1.4 Question A - Part 4: Automate Test Packets

This module is the last piece of question A. It should use the already parsed rules from **violation_checker** and create packets and conditions that would violate these rules. For instance if **H1** should not communicate with **H2**, create packets from **H1** to **H2** in the controller, send it and see if **sdn_dump** captures these packets. If the packets get caught with a reply from a host or forwarded through the network by **violation_checker**, then we have produced policy violating behaviour that the description of the the intended behaviour says is not allowed, but that we can show works on the actual network. This means that they don't match and that we should continue our search down to Question B. While inspired from ATPG[1] the difference here is that we are creating the packets from the policy description itself, and not from forwarding rules on the network devices. We are ultimately just doing static checking on certain rules the user wants to test on a live network.

When the **automator** starts it will be given each parsed policy rule by **violation_checker**. The **automator** will wait until the last network switch has turned on and then it will traverse each of these policy rules and attempt to create violations by creating a matching packet for that rule and then send it out to the network. It is important to mark this packet with an action called *(port=OFPP_CONTROLLER=)*[12] so that all other control applications on the network that are listening for new incoming packets also receive this packet, essentially we are sending the packet back to the controller through the last switch that turns on.

Listing 4.6: Automate Packet Creation

```
1 def create_packets():
2   for rule in policy_rules:
3     if some_protocol_type in rule:
4           packet = create_packet_type(rule.src, rule.
              dst)
5           send_packets(packet)
```

During the creation of the packets the **automator** should first look at what protocol type the rule is describing. It should always choose the simpler version possible of the rule. That would mean that a rule doesn't have a prototype constraint in it e.g. **Time 10:00 to 12:00 block H1**, should be tested with an ICMP rather than a TCP.

Note that the **automator** can only test Time and Date rules since they are directly packet related, which means that Data rules are rather a check on the network done by **sdn_dump** to see if certain conditions are true on each network device.

Question A - Part 4: Summary

- Learns the number of switches
- Learns the policy rules
- Once the last switch has started it will create packets
- Create packets by using the parsed policy rule
- Ensure that the controller also receives the sent packet

4.1.5 Question A - Model Overview

This subsection aims at showing the overall model and data flow of Question A: Design. The UML **(Figure 4.1)** shows that Question A will have three major classes **sdn_dump** (the packet listener), **violation_checker** (the packet checker) and **automator** (the packet generator). In Question A flow digram **(Figure 4.2)** we can observe the designed flow. Note that we have the ability for the user to leave the **automator** or the **automator** and **violation_chcker** from running using commands to **sdn_dump**. This gives the program multiple functionalities such as: just a packet listener, listener and checker and a full automation of this equivalence check.

4.1.6 Question A: Next Step

As stated previously, if a violation is found then **sdn_dump** will be alerted, note what packets and rule it was, then print out first part of the error stack. Then **sdn_dump** should start the next step in the decision tree (**Figure** 2.4) by troubleshooting on Question B: "Does the Device State Match the Policy?"[2]. We can do this by keeping a boolean value for each check we do, and if it return True on a violation check, it should mean that we need to move downward the tree.

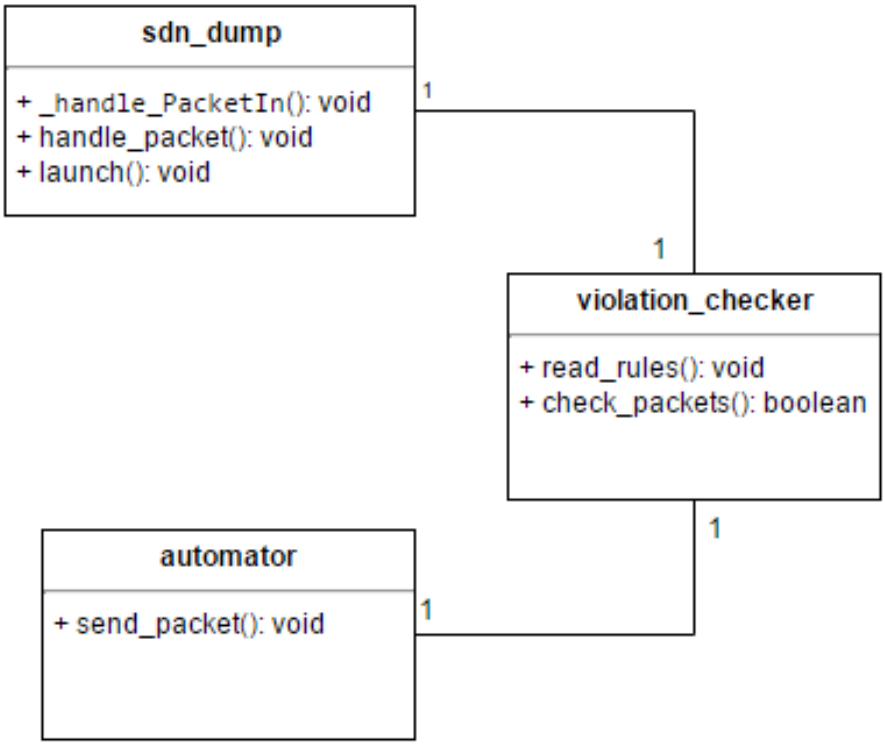

Figure 4.1: UML: Class Overview

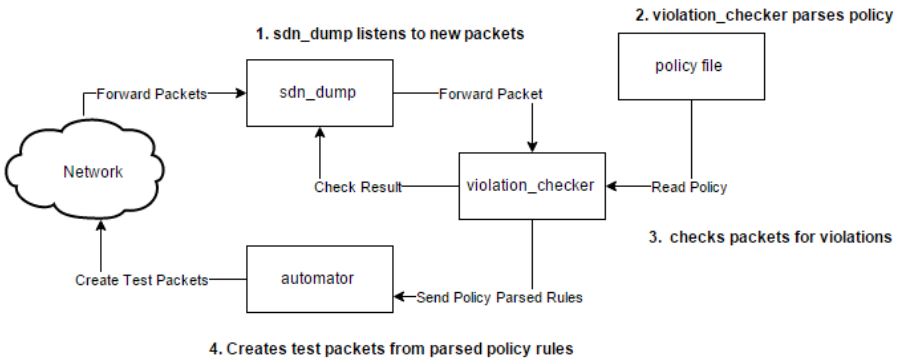

Figure 4.2: Model Data Flow

4.2 Question B: Design

Once we have found a violation on Question A: "Does policy match actual behaviour?", we need to move on to Question B: "Does policy match device state?". The important issues to find here are network errors such as forwarding loops, black holes and disconnectivity issues[2]. Once we are here we also need to facilitate a way to move further down the decision tree once we have answer to if there is forwarding loops, black holes or disconnected devices.

4.2.1 Question B: Forwarding loops

These forwarding loops occur when we have multiple connections between two network switches or when two ports on one switch are connected to each other[28], and this has the potential to cause a circular network loop[29]. Since there is no TTL, an Ethernet packet has the potential to loop forever, at least until dropped due to resource exhaustion[28]. We will thereby use a spanning tree protocol to allow redundant/multiple links, without network failure[30].

4.2.2 Question B: Disconnectivity issues

After we have started the spanning tree protocol, we need to check disconnected switches. Since OpenFlow keeps track of all its device connections, we can count them and compare that to a number of switches we are expecting the tool to find. If the numbers mismatch, we need to inform the operator and act accordingly on the decision tree.

4.2.3 Question B: Move down the tree

From Question B we can move to Question C (No) and Question D (Yes) based on the answer to the question. So before we answer Question B we should know if the network is experiencing looping behaviour, black holes or/and have disconnected devices.

4.3 Question D: Design

When we reach Question D, we are at a leaf of the decision tree, meaning that we are at an endpoint. To recap Question D is "Does the device state match the hardware?", Heller states that if *Yes*, then buggy hardware is to blame, if *No*, then firmware is the cause[2]. This can be seen in the forwarding table of the network devices. Either way we request the forwarding tables from the devices and filter out flow entries based on the last violating packet we found. The end goal here is for the operator to know what policy rule, network device, ports and flows are creating a network problem. Since we then already know the policy rule and packet that are a policy violation, we can extrapolate the flows, and thereby also answer to Question D. This means if we find flows that are corresponding to the policy rule and violating packet, then we have *No* answer something is wrong in the configuration. If no flows are found to match then it would be a Hardware problem. From here, human intervention would be needed to fully solve the problem.

4.3.1 Question D: Request flow entries

With OpenFlow, we have the ability to know all the flow statistics from each network device. This is called an *ofp_flow_stat_request*[12, 13] which returns valuable information about the flow table of each device. Each

entry contains values such as packet count, byte count, a timer, a match, actions and much more. The values we are interested in both for debugging and functionality are the byte count, match and action.

4.3.2 Question D: Filter out data

In the section above we stated the need for three values from the flow entries. We use a filter on these values to discern flow entries related to our policy rule and packet violation, and then present them to the user. After the filtering the user should have a clear picture of what is causing an error across the network, both on the high level policy and low level configuration.

Byte count value

This value is more important when the Data policy rule has been violated, but has its uses on the other rules too. With it we can show the operator how much each flow is transmitting. We should show this value for each flow entry related to the violation.

Match value

The match value varies based on what protocol the flow entry is concerning. E.g. an ICMP flow entry will not have values such as source and destination ports, while a TCP flow would have. By remembering the last violating packet we can match its values such as protocol, source, destination and port vs. the flows protocol, source and destination. The match also gives us an important value called **in_port** which tells us what entry port the flow has. This is the first step in giving the user a port set (**in** and **out port**) on the network device.

4.3.3 Actions

The action value tells us what action to take if a packet matches the flow match, and on what port to do the action. The most normal action we see is the *OFPAT_OUTPUT* which says that if a packet match that flow go out on this switch port[13]. By using this information we inform the operator of the action and corresponding port.

4.3.4 Presenting the information

Once we have filtered out the packet, we need to show the user the violating **flow match** and **in-port**, corresponding **action** and **out-port**, and **byte count** for each network device. The user now not only knows the violating packet path (from one device to another device until destination), but also has a path of corresponding flow entries for each jump in the path showing him/her what is wrong with each low level configuration on each device. The expected result would look like as shown in **Figure 4.3**

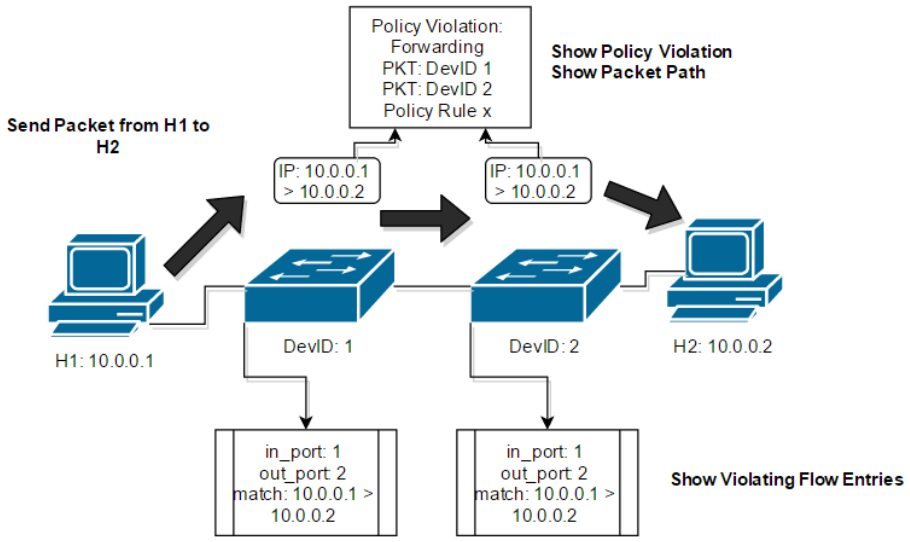

Figure 4.3: How violating behaviour will be shown

Chapter 5

Implementation Results

This chapter will show implementation of the prototype from the design in the previous chapter. The section is mainly dived into three parts for each of Question (A, B, C). This chapter aims at explaining implementation choices done on the prototype. Snippets of code will also be shown, the code in its full length will be listed in the appendix C and GitHub repository **https://github.com/HarisSistek/public_sdntroubleshoot**.

5.1 Helpful mail archives, sources and wikis

I used a number of helpful guides, sources, mail archives etc. during this implementation that I would like to reference. These two archived mailing correspondences were very helpful in understanding how to create ICMP packets[31] and TCP packets[32] with the POX API. Further the POX wiki[12] and NOX/POX repo[33] were instrumental in understanding how events and event listeners worked and could be used.

5.2 Question A: Implementation

The code is divided into three main files, with an additional directory "**policies**" for the policy files.

5.2.1 Files and Directory

The tree structure of the tool directory:

Listing 5.1: Directory tree structure

```
1 .
2 |-- __init__.py
3 |-- policies
4 |   |-- policy1.pol
5 |-- automator.py
6 |-- sdn_dump.py
7 |-- violation_checker.py
```

5.2.2 How to start the controller app

The application uses POX to run, and connects to the Mininet topology if the user has specified the wish for a remote controller.

Listing 5.2: Mininet example

```
1 sudo mn --topo single,3 --controller remote
```

After the topology is running we can use POX to run the app on the network by:

Listing 5.3: Example executions

```
1 ./pox.py pox.sdntroubleshoot.sdn_dump --switch=1 --
    mode=2
2 ./pox.py pox.forwarding.l2_learning pox.
    sdntroubleshoot.sdn_dump --switch=1 --mode=3
```

From here the listener at **sdn_dump.py** will start to monitor, check and create network behaviour based on what the policy description says.

Sdn_dump parameters

Sdn_dump needs two parameters to start, the *switch* and *mode*. *Switch* signals the number of network devices and *mode* sets operation mode for the tool.

Listing 5.4: Tool Parameters

```
1 --switch=2 --mode=1 # Two switches. Listen for packets
    .
2 --switch=1 --mode=2 # One switch. Listen and check
    packets.
3 --switch=1 --mode=3 # One switch. Listen, check and
    create packets.
```

5.2.3 Question A - Part 1: Capture Actual Network Behaviour

Sdn_dump.py is the main file of the app. It will listen to packets, start the checking of packets and decide when to move down the decision tree for Question B.

The launch

Sdn_dump's launch() function will initiate everything. We start by giving this function two variables from the command line, switch and mode. Switch is used in order for the program to know how many network devices to wait for before starting the creation of packets, as this number is also used the number during the connectivity check during Question B. Mode is used by the code to know what operation mode it is in. The three modes are listen, listen and check, and lastly listen, check and automate.

Listing 5.5: Sdn_dump.py launch()

```
1  def launch (switch = "", mode= ""):
2    global checker
3    global switch_count
4    print "Number of switches expected:", switch
5    switch_count = int(switch)
6    start_spanning_tree()
7    if "2" in mode or "3" in mode:
8        checker = vc.Violation_Checker(switch, mode)
9    core.openflow.addListenerByName("PacketIn",
       _handle_PacketIn)
10   core.openflow.addListenerByName("PortStatsReceived",
         handle_port_stats)
11   core.openflow.addListenerByName("FlowStatsReceived",
         handle_flow_requests)
```

We also define what event we are listening to and what function should handle each event type. **Sdn_dump.py** has one handler for each of the three events listed in the bottom of the code snippet. Subsequently, we should now see how **_handle_PacketIn(event)** deals with new packets coming to the controller.

Listening mechanic

Once the adder has marked **_handle_PacketIn()** to the *PacketIn* events we can create a network wide tcpdump. All packets that we receive will be parsed and printed out on terminal. Since there are many different packet and protocol types we first have to differentiate between them. **_handle_PacketIn()** does the first step of packet differentiation by:

Listing 5.6: _handle_PacketIn()

```
1  def _handle_PacketIn(event):
2    global count
3    packet = event.parsed
4    if packet.find("arp"):
5      handle_arp(dpid_to_str(event.dpid),packet)
6    elif packet.find("ipv4"):
7      handle_ip(dpid_to_str(event.dpid),packet)
8    else:
9      log.info("UNKNOWN packet %s", packet.src)
10
11   count = count + 1
12   if count == 5: # every 5 packets request port data
13     send_requests()
14     count  = 0
```

We see that **_handle_PacketIn(event)** takes the *PacketIn* event, parses the event into the packet. Once we have this packet we distinguish between

it being an ARP packet and an IP packet. There is also a count variable there we use to request port statistics for each 5 packets we have received from the network. Since **handle_arp** and **handle_ip** are similar functions, we can show the most relevant one.

Listing 5.7: handle_ip()

```
1  def handle_ip(dev_id,packet):
2    ip_packet = packet.payload
3    srcip = ip_packet.srcip
4    dstip = ip_packet.dstip
5
6    add_host(dev_id, port, srcip)
7
8    if ip_packet.find("icmp"):
9      handle_icmp(dev_id, packet, ip_packet, srcip,
         dstip)
10   elif ip_packet.find("tcp"):
11     handle_tcp(dev_id, packet, ip_packet, srcip, dstip
         )
12   elif ip_packet.find("udp"):
13     handle_udp(dev_id, packet, ip_packet, srcip, dstip
         )
```

Handle_ip() will take a the *event.id* (meaning the ID of the device that created the event) and received packet. From there it will extract packet payload, and the source and destination IP. By looking at the packet payload we find what kind of protocol the packet is using and forward it to the appropriate function. We also use the function **add_host()** to map *IP-PORT-DEVICE* that will be used and covered in a later subsection.

Handling protocols

Once an IP packet has been parsed it will be sent to one of the three possible handle functions: **handle_icmp()**, **handle_tcp** and **handle_udp**. Again these functions are very similar and covering one of them should be enough to understand the implementation of the other two.

Handle_udp() will take device ID, the received packet, the packet payload (ip_packet), source and destination address. From the function will:

Listing 5.8: handle_udp()

```
1  def handle_udp(dev_id, packet, ip_packet, srcip, dstip
     ):
2    udp  = ip_packet.payload
3    srcport = udp.srcport
4    dstport = udp.dstport
5    x =  "%s␣DevID:␣%s␣UDP:␣pkt␣%s␣>␣%s,␣ip␣%s␣>␣%s:␣
       srcp␣%s␣dstp␣%s" % (timestamp(), dev_id, packet.
       src, packet.dst, srcip, dstip, srcport, dstport)
```

```
6
7   if layer1_correct:
8     print x
9     if checker:
10        violation = checker.check_if_legal(x)
11        decider(violation, x)
```

From there it will extract the payload from the IP packet. This payload (udp) contains the source and destination ports used. From there we create the string "x" that we then print out and use to check against the network policy with the checker if we still hold that layer1 or Question A is correct. This means that we will only print out or check network packet while we don't have an answer for Question A.

Forward packet for checking

We check packets by using the *checker* and the function **checker.check_if_legal()** as seen in the code above. Once the function returns a boolean, we will use it to gauge the next decision for the program, namely keep listening or move down the decision tree? The function **decider()** that receives this boolean value from **violation_checker** will be covered during Question B: Implementation.

Mapping IP-PORT-DEVICE

We map this relationship using the **add_host()** function. The functions start by checking if we already have the IP inside the dictionary, as we don't want to map the same IP address twice.

Listing 5.9: add_host()

```
1  def add_host(dev,port,ip):
2      global dev_port_ip
3      global known_maps
4      if ip in known_maps:
5          return
6
7      ... update host ...
8
9      info = {}
10     info[ip] = port
11     info[port] = ip
12     dev_port_ip[dev] = info
13     known_maps.append(ip) # just map once
```

Then we map the IP to a port, and port to an IP for that device ID. We finally add the IP to known maps so we don't map this same relationship again.

Request port statistics

We covered during **_handlePacketIn()** function that every 5 packets we request port data. The function **send_request()** covers this by simply looping trough the network device connections that OpenFlow keeps and sends a request message for the port data.

Listing 5.10: send_request()

```
1  def send_requests():
2      for con in core.openflow._connections.values():
3          con.send(of.ofp_stats_request(body=of.
               ofp_port_stats_request()))
```

After the requests are sent we wait for the listener **handle_port_stats()** to catch the event.

Handle port statistics

Handle_port_stats() will handle the port statistics sent back to the controller. We use it to update how many packets and bytes each IP has sent and received.

Listing 5.11: handle_port_stats()

```
1  def handle_port_stats(event):
2      stats = event.stats
3      for stat in stats:
4        if dev_port_ip.get(dpid_to_str(event.dpid)):# if
             entry exists
5          if dev_port_ip.get(dpid_to_str(event.dpid)).
               get(stat.port_no):# if entry exists
6            add_port_entry(dev_port_ip[dpid_to_str(event
                 .dpid)][stat.port_no], stat.tx_bytes,
                 stat.rx_bytes, stat.tx_packets, stat.
                 rx_packets)
7      if layer1_correct:
8          if checker:
9              violation = checker.check_if_ports_legal(
                   ip_bytes_sent, ip_bytes_recv,
                   ip_packets_sent, ip_packets_recv)
10             decider(violation, None)
```

We first extract the stats from the event and if we already have mapped this device, port and IP in the *IP-PORT-DEVICE* relationship, we keep and update that stored relationship with the new port stats using **add_port_entry()**. Again if we are still looking for a violation to layer1 or Question A, we check the new port data against the rules with **violation_checker**, and take the response to the decider to see what to do next.

5.2.4 Question A - Part 2: Describe Intended Behaviour

This section covers how the policy is read, parsed, stored and how it will be interpreted by the program. These things are handled by the **Violation_Checker** class **violation_checker.py**. We can start by explaining the key variables that **violation_checker.py** (hereby referred to as **VC**). **VC** keeps track of a few things from the **init()** phase; policy hostnames and IP, the relationship between the two, the policy rules in string type and the same policy rule parsed and stored into python dictionaries.

Reading policy files

Once the correct variables have been created, the **VC init()** will call **read_policy_folder()** function that will read all the policy files in the **policies** directory. The function reads every file with *.pol* extension in the directory and for each line in the file it will distinguish between host declaration line or policy rule, and will ignore comment lines.

Listing 5.12: read_policy_folder()

```
1  def read_policy_folder(self, policy_path):
2      ...
3          for line in lines:
4              if re.match(r'^.+\s+=\s+.+',line): #
                   interpret hosts
5                  match = re.match(r'^(.+)\s+=\s+(.+)',line
                       )
6                  self.policy_hosts_ip[match.group(2)] =
                       match.group(1)
7                  elf.policy_hosts_name[match.group(1)] =
                       match.group(2)
8              elif re.match(r'^Date',line) or re.match(r'
                   Time',line) or re.match(r'^Vlan',line) or
                    re.match(r'^Data',line):
9                  self.policy_rules.append(line)
10                 self.policy_rules_values.append(self.
                       interpret_block_and_options(line))
```

The host declarations will be mapped into the appropriate dictionaries, it will be bidirectional so that we can extract the hostname from IP and vice-versa. If the line is a policy rule, it will both store the string and store the parsed version of the rule by calling the **interpret_bloc_and_options()** function that will parse each rule according to its type.

The flow of interpreting policy rules

We can visualize the parsing of each line read by **read_policy_folder()** through the use of a flow chart such as that on **(Figure 5.1)**

Once the program reaches one of the end **interpret_*ruletype*_rule()** it will return its result back down to **read_policy_folder()** which will append

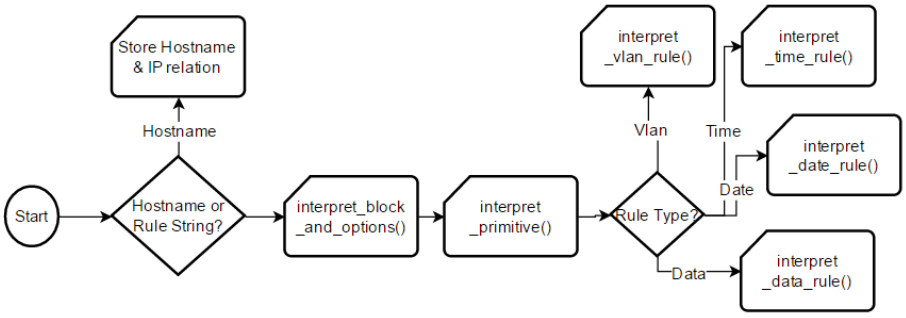

Figure 5.1: Flow for parsing each rule

the parsed rule to the stored list we keep, *policy_rules_values*.

Interpreting Time policy rules

Since Time, Date and Vlan rules are parsed and stored very similarly it will be enough to cover how Time is parsed, and thereby also explaining the method of parsing for the other two. We can see the structure of the Time policy rule again for reference:

Listing 5.13: Time rule examples

```
1 Time 10:00 to 23:00 block host1 to host2 prot TCP
     sport 4000 dport 5001
2 Time 09:30 to 20:00 block host1 to host2 prot UDP
3 Time 10:00 to 23:00 block h2 prot TCP
4 Time 10:00 to 23:00 block h3 to 10.0.0.3
```

Once **interpret_bloc_and_options()** receives policy rule line it will parse it at different stages. The stages are:

Decide protocol type:

The first thing the function does is check what protocol the rule may be referring too, i.e. what has been specified in the rule? The function accounts for many different protocol types (TCP, UDP, ARP, ICMP) and even some self defined ones for tool specific uses.

Listing 5.14: interpret_bloc_and_options()

```
1 def interpret_block_and_options(self, rule):
2     prot = {}
3     if "prot TCP" in rule:
4         prot["prot"] = "TCP"
5         return self.interpret_primitive(prot, rule)
6     ...
7     elif "Data" in rule:
8         prot["prot"] = "Data"
9         return self.interpret_primitive(prot, rule)
```

```
10      else:
11          prot["prot"] = "TCP/UDP/ICMP"
12       return self.interpret_primitive(prot, rule)
```

Specifying protocol type is most important for the Time and Date rule. Because we are testing packets and later creating packets, we need to know what protocol to use when **automator.py** creates new packets and what packets to filter out when **VC** test packets for violations. Also note that if no protocol is specified in a Time or Date rule, we assume that the rule is referring to ICMP, TCP and UDP packets, and should trigger at each one of them.

Decide rule type:

The last thing **interpret_bloc_and_options()** does before returning is call the second function in the parsing logic. Calling **interpret_primitive()** will distinguish between the different rule types/primitives such as Time, Date, Vlan and Data, and send them to type/primitive specific rule functions that will interpret such rules.

Listing 5.15: interpret_bloc_and_options()

```
1  def interpret_primitive(self, prot, rule):
2      if "Time" in rule:
3          return self.interpret_time_rule(prot,rule)
4      ...
5      elif "Data" in rule:
6          return self.interpret_data_rule(prot,rule)
7      else:
8          pass
```

Parse Time policy rule:

The **interpret_time_rule()** function is the second last step in parsing a Time rule. It will firstly create a dictionary and store both the rule type (Time) and the policy rule string. After that we will need two regexe matches; one that specifies connection between two hosts (Time rule 1,2 and 4 in the Listing 5.8 above) and one that just specifies a host to block (Time rule 3 in the same Listing).

Listing 5.16: interpret_time_rule()

```
1  def interpret_time_rule(self,prot,rule):
2      ret_dict = {}
3      ret_dict["rule_type"] = "Time"
4      ret_dict["rule_string"] = rule
5      match = re.search(r'^Time\s+(?P<start_time>.+)\s+
           to\s+(?P<end_time>.+)\s+block\s+(?P<from>.+)\s+
           to\s+(?P<to>\S+)\s+', rule)
6      if match:
```

```
7         options = self.interpret_options(rule)
8         ret_dict = dict(list(ret_dict.items()) + list(
              match.groupdict().items()))
9         return self.merge_dicts(prot, ret_dict, options)
10      else:
11        match = re.search(r'^Time\s+(?P<start_time>.+)\s
              +to\s+(?P<end_time>.+)\s+block\s+(?P<from>\S
              +)\s+', rule)
12        if match:
13          options = self.interpret_options(rule)
14          ret_dict = dict(list(ret_dict.items()) + list(
                match.groupdict().items()))
15          return self.merge_dicts(prot, ret_dict,
                options)
16        else:
17          return None
```

As we can see in the code above, the function will extract values such as time interval (start and end), and specify if it is a connection we want blocked or just a host. From there we need to extract additional values for TCP and UDP related rules, this is covered in the next sub section.

Parse Time rule options:

You can also note that we use a method **interpret_options()**. The functions will read the line and look for UDP and TCP specific options, this means ports and flags.

Listing 5.17: interpret_options()

```
1  def interpret_options(self, rule):
2      match = re.search(r'sport\s+(?P<sport>.+)\s+dport\
           s+(?P<dport>.+)', rule)
3      if match:
4        return match.groupdict()
5      else:
6          ...
```

Once the options are found, we can merge the three different dictionaries we have prot (stored the rule protocol), *ret_dict* (stored the rule type specific data) and options (stored TCP and UDP specific data, such as ports). Lastly we merge these three dictionaries with **merge_dicts()** which was created to merge three dictionaries into one.

Listing 5.18: merge_dicts()

```
1  def merge_dicts(self,prot,type_dict,options):
2      ret_dict = {}
3      if options:
4        ret_dict = dict(list(prot.items()) + list(
             type_dict.items()) + list(options.items()))
```

```
5       else:
6         ret_dict = dict(list(prot.items()) + list(
              type_dict.items()))
7       return ret_dict
```

Once the dictionary has been merged we return it down the many returns back to the read **policy_file_folder() (read_policy_folder <- interpret_block_and_options <- interpret_primitve <- interpret_time_rule)** function, and append the parsed policy rule to our policy_rules_values list that stores all the different parsed policy rules.

Interpreting and Parse Data policy rules

Data rules are bit different then the other ones, here we need to convert the limit and size notation into number of bytes. We need this because port statistic data from OpenFlow *ofp_port_stats* message[13] is returned in bytes and not KB, MB and greater size notations. Everything will be similar until we reach the **interpret_data_rule()**. To recap data rules look like:

Listing 5.19: Data policy rules

```
1 Data 1 KB to host1
2 Data 1000 b from host2
3 Data 10 KB to 10.0.0.100
```

Where **"to host1"** means how much data **host1** can receive and **"from host2"** means how much data **host2** can send. The parse function for data rules work by extraction values such as data limit, data notation type and from/to.

Listing 5.20: interpret_data_rules()

```
1  def interpret_data_rule(self,prot,rule):
2      ret_dict = {}
3      ret_dict["rule_type"] = "Data"
4      ret_dict["rule_string"] = rule
5      match = re.search(r'^Data\s+(?P<lim>.+)\s+(?P<
           notation>.+)\s+from\s+(?P<from>.+)',rule)
6      if match:
7        ret_dict = dict(list(ret_dict.items()) + list(
             match.groupdict().items()))
8        return self.merge_dicts(prot, ret_dict, {})
9      else:
10       match = re.search(r'^Data\s+(?P<lim>.+)\s+(?P<
             notation>.+)\s+to\s+(?P<to>.+)',rule)
11       if match:
12         ret_dict = dict(list(ret_dict.items()) + list(
               match.groupdict().items()))
13         return self.merge_dicts(prot, ret_dict, {})
14       else:
15         return None
```

As in the **interpret_time_rule()** the function will return a merged dictionary all the way down to the **read_policy_folder()** and append the newly parsed rule into the *policy_rules_values* list.

5.2.5 Question A - Part 3: Match Actual Behaviour against Intended behaviour

Once the policy rules have been interpreted and stored into python dictionaries we can use them to match rules against live network behaviour. This would in turn mean matching rules against packets.

Parsing network packets

The first step in matching is to parse a packet into usable data, and we do this trough the **check_if_legal()** function. The function has a series of regexe expressions for different packet types, e.g. ICMP packet.

Listing 5.21: ICMP packet parsing in check_if_legal()

```
1 def check_if_legal(self,dump_string):
2     if "ICMP" in dump_string:
3         match = re.match(r'^.+\s+DevID:\s+(?P<devID>.+)\
              s+(?P<prot>.+):\s+pkt\s+(?P<pktsrc>.+)\s+>\s
              +(?P<pktdst>.+),\s+ip\s+(?P<ipsrc>.+)\s+>\s
              +(?P<ipdst>.+):\s+(?P<ping_type>.+)',
              dump_string)
4         match_dict = match.groupdict()
5         violation = self.check_icmp(match_dict,
              dump_string)
```

Using this method we can also parse the other packet types we need to match (UDP, TCP and ARP). Once this is done we send them to the next appropriate function. These functions are **check_icmp(), check_udp() and check_tcp()**, which we choose based on what packet type/protocol the **check_if_legal()** function received.

Flow chart for checking packets against rules

From **check_if_legal()** the flow on **(Figure 5.2)** follows.

We see on the flow chart that each packet tested has two results *"True"* or *"False"*. Either the packet will violate a rule or it will not.

Match packet against rules

When the packet has been parsed and sent to one of the three appropriate functions, we start to match the packet against all the stored rules. Since the three functions are very similar we will show how **check_udp()** works since it covers all of the main issues of matching packets. In matching a packet against a rule we need to cover 4 things. For each rule we have stored we need to:

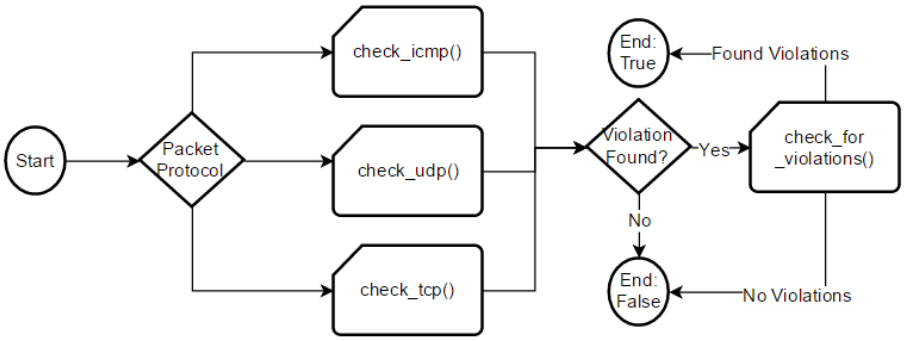

Figure 5.2: Flow for checking packets

1. Extract correct IP from hostnames
2. Compare if packet IPs match the rule IPs
3. Compare if packet ports match the rule ports
4. Check if packet is inside the rules time or date interval

Step 1: Extract IP from hostnames

In the code snippet we can see the function first distinguishes each rule to the correct protocol, and then we extract the correct IPs.

Listing 5.22: Step 1: check_udp()

```
1  def check_udp(self, packet_dict, packet_string):
2      for rule_dict in self.policy_rules_values:
3          if "UDP" in rule_dict["prot"]: # rule may miss
               these keys, therefor use get -> will return
               None if key is non existent
4              r_src = None
5              r_dst = None
6              match = re.match(r'\d{1,3}\.\d{1,3}\.\d
                   {1,3}\.\d{1,3}', rule_dict.get("from"))
7              if match:
8                  r_src = rule_dict.get("from")
9              else:
10                 r_src = self.policy_hosts_name.get(rule_dict
                       .get("from"))
11             if rule_dict.get("to"):
12                 match = re.match(r'\d{1,3}\.\d{1,3}\.\d
                       {1,3}\.\d{1,3}', rule_dict.get("to"))
13                 if match:
14                     r_dst = rule_dict.get("to")
15                 else:
16                     r_dst = self.policy_hosts_name.get(
                           rule_dict.get("to"))
```

If the policy rule contains an IP we use that, but if we see a hostname we extract it from our stored hostname to the IP dictionary *(policy_hosts_name)* we created from reading the policy file.

Step 2: Compare packet IP vs. rule IP

Once we have the correct IP value for rule source and/or destination (destination is optional in writing rules), we can match them against that of the packet. This is done by the **check_rule_and_packet()** function.

Listing 5.23: check_rule_and_packet()

```
1 def check_rule_and_packet(self, r_src, p_src, r_dst,
     p_dst):
2     ret1 = self.check_two_values(r_src, p_src)
3     ret2 = self.check_two_values(r_dst, p_dst)
4     if ret1 and ret2:
5       return ret1 and ret2
6     else:
7       ret1 = self.check_two_values(r_dst, p_src)
8       ret2 = self.check_two_values(r_src, p_dst)
9     return ret1 and ret2
```

This function has a somewhat complicated logic, but I will try to explain it below. We can have two situations one the packet matches the rule completely. Rule source matches packet source, and rule destination matches packet destination. However, this covers the connections communication only one way, what about the reply to the communication? In order to accomplish, we also need to match where rule destination matches packet source and rule source matches packet destination. We also have to remember that rule source and destination can be a **"*"-value**, meaning that it can be anything, and rule destination can also be a null value, meaning we have specified a communication between two host, but rather just blocked one host (source). We handle this by using **check_two_values()** function:

Listing 5.24: check_two_values()

```
1 def check_two_values(self, rule_val, packet_val):
2     if rule_val == "*" or rule_val == None:
3       return True
4     else:
5       return rule_val == packet_val
```

Both of these functions return a boolean value to the caller, and the caller (this case: **check_rule_and_packet()**) will check ports. If *True* or check the next stored rule if *False*.

Listing 5.25: Step 2 and 3: In check_udp()

```
1 if self.check_rule_and_packet(r_src, packet_dict["
     ipsrc"], r_dst, packet_dict["ipdst"]):
```

```
2     if self.check_ports(rule_dict.get("sport"),
          packet_dict["src_port"], rule_dict.get("dport"),
           packet_dict["dst_port"]):
```

Step 3: Compare packet ports vs. rule ports

The next function we need to use is one that will compare packet and rule ports. The function is called **check_ports()**:

Listing 5.26: check_ports()

```
1 def check_ports(self, r_sport, p_sport, r_dport,
      p_dport):
2     ret1 = self.check_two_values(r_sport, p_sport)
3     ret2 = self.check_two_values(r_dport, p_dport)
4     return ret1 and ret2
```

You will notice that while in **check_rule_and_packet()** we had to compare these two in two possible situations; the request and reply. Here, for the sake of simplicity, we will only check if rule and packet source/destination port match and not cover the other side of the communication of this connection. This means that the user has to specify this with two different rules if he wants to cover both way communication when also matching a connection with ports specified in the rule. Example:

Listing 5.27: check_ports()

```
1 Time 10:00 to 23:00 block H1 to H2 prot TCP sport
      41238 dport 5001
2 Time 10:00 to 23:00 block H2 to H1 prot TCP sport 5001
     dport 41238
```

Step 4: Check time and date intervals

Once we have covered all the similar rule properties such as IPs and ports, we can look at what differentiates the rules, by taking the time time or date interval and check if the packet time is between the two ends (**start - end**). These to rule types and how we handle them are very similar, so for convenience we will only cover the Time-rule.

First we need to distinguish the rule between Time or Date, this is done by **check_time_or_date()** function.

Listing 5.28: check_time_or_date()

```
1 def check_time_or_date(self, rule, packet,
      packet_string):
2     match = re.search(r'^(?P<date>\S+)\s+(?P<time>\S+)\
         s+', packet_string) # extract date and time from
          packet timestamp
```

```
3     if rule["rule_type"] == "Time":
4         if match:
5             mdict = match.groupdict()
6             packet_time = time.strptime(mdict.get("time")
                  ,"%H:%M:%S")
7             rule_start_time = time.strptime(rule.get("
                  start_time"),"%H:%M")
8             rule_end_time = time.strptime(rule.get("
                  end_time"),"%H:%M")
9
10            if packet_time >= rule_start_time and
                  packet_time <= rule_end_time:
11                return True
12            else:
13                return False
14    elif rule["rule_type"] == "Date":
15        ...
```

Then we extract the day and timestamp from the packet string. Using this we convert both the time-rule interval and packet time into python time objects. Now we have a way of comparing if the packet violates a time or date rule. This takes us to the last step in checking if a packet violates a rule.

Listing 5.29: Step 4: In check_udp()

```
1     ret = self.check_time_or_date(rule_dict,
          packet_dict, packet_string)
2
3     if ret:
4         return rule_dict["rule_string"]
```

The code snippet above shows that we return the rule text if we find a violation. We use this later in **check_if_legal()** to see if a true violation is found, meaning the packet has been forwarded or replied to. Lastly, we mark lastly mark this packet in the violation list *(v_packet)*, and finally call the function **check_for_violation()** to discover if this packet has created a violation on the network.

Listing 5.30: Last step in check_if_legal()

```
1 if violation:
2     string_dict = {}
3     string_dict["packet_string"] = dump_string
4     rule_dict = {}
5     rule_dict["rule_string"] = violation
6     res_dict = self.merge_dicts(match_dict, string_dict
          , rule_dict)
7     self.v_packets.append(res_dict)
8     if self.check_for_violation(res_dict):
9         return True
```

```
10 return False
```

You can also note that this will return a True boolean or False to **sdn_dump.py** based on what it discovers in **check_for_violation()**.

Discovering a violation

To find a violation we have captured and marked 2 packets for violation. A violation would be that a marked packet has been forwarded from one network device to another, or that we see two marked packets on the same device, a request and reply. We solve this by using the function **check_for_violations()**, which receives dictionary of the last violated packet we found and marked. We use that dictionary to check it against all other marked packets and look for a relation. We also need to note if the packet is an ICMP or TCP/UDP. This would involve two different cases, one where we look at ICMP reply code, and one where we look at the port numbers. For simplification we will cover how we find an ICMP violation. First we loop through all the marked packets and look for where the marked packet we sent in as a parameter and the packet in the list has equal protocol.

Listing 5.31: Step 1: check_for_violation()

```
1 def check_for_violation(self, packet_dict):
2     for vpacket in self.v_packets: # for violating
          packet in violation list
3         if vpacket["prot"] == packet_dict["prot"]:
4             ...
```

Finding a violating reply

A violating reply is found when we see a marked reply packet on the same device as we have marked a request packet, but the reply packet should have switched reply code and IPs (**Figure** 5.3).

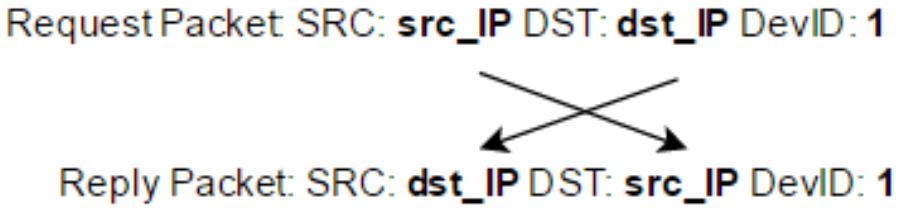

Figure 5.3: Discovering a reply violation

We can see this done in the code listing, for each marked packet we check the devID matches with our last marked packet. Then we ensure that the IPs are switched and finally we ensure that this is a reply to a marked packet.

Listing 5.32: Step 2.1: check_for_violation() Find Reply

```
1  if vpacket["devID"] == packet_dict["devID"]: # packet
       reply on same device
2      if vpacket["ipsrc"] == packet_dict["ipdst"] and
           vpacket["ipdst"] == packet_dict["ipsrc"]: # is
           it a reply
3          if vpacket["prot"] == "ICMP":
4              if "8" in vpacket["ping_type"] and "0" in
                   packet_dict["ping_type"]: # ping type is a
                    reply
5                  ... print output ...
6                  if vpacket in self.v_packets:
7                      self.v_packets.remove(vpacket)
8                  if packet_dict in self.v_packets:
9                      self.v_packets.remove(packet_dict)
10                 return True
11         else:
12             ... UDP/TCP ...
13 else:
14     ... different device, check forwarding ...
```

Once we have found a Reply Violation we print out the data, as shown in Listing 5.33:

Listing 5.33: Reply Violation Found

```
1  ## Policy Violation Found: PACKET RETURN ##
2  > Packet: Sun 07:29:52 DevID: 00-00-00-00-00-01 ICMP:
       pkt ff:ff:ff:ff:ff:ff > ff:ff:ff:ff:ff:ff, ip
       10.0.0.1 > 10.0.0.2: 8:ECHO_REQUEST
3  > Packet: Sun 07:29:52 DevID: 00-00-00-00-00-01 ICMP:
       pkt ae:7e:27:15:9b:98 > 3e:8b:5d:65:11:d9, ip
       10.0.0.2 > 10.0.0.1: 0:ECHO_REPY
4    violates:
5  > Rule: Date MON to SUN block host1 to host2
```

We want to keep the list small and don't want to fire a violation one the same two packets again, so we remove these two packets from the list with marked packets *(v_packets)*. Finally we return a *True* or *False* boolean based on what the function finds, which will make the function caller signify **sdn_dump.py** about the violation.

Finding a violating forwarding

A forwarding violation is found if we see the same packet, but on two different devices (**Figure** 5.4).

Listing 5.34: Step 2.2: check_for_violation() Find Forwarding

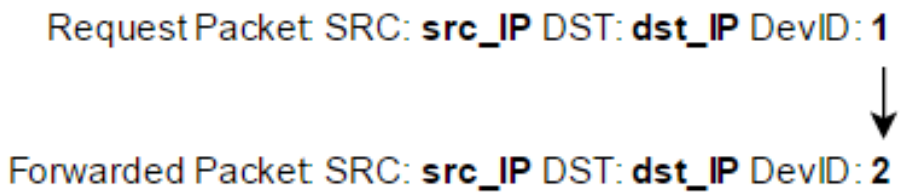

Figure 5.4: Discovering a forwarding violation

```
1  if vpacket["ipsrc"] == packet_dict["ipsrc"] and
       vpacket["ipdst"] == packet_dict["ipdst"]: # packet
       has been forwarded
2      if vpacket["prot"] == "ICMP":
3          if vpacket["ping_type"] == packet_dict["
               ping_type"]: # ping type is a reply of
               another
4              ... print output ...
5              if vpacket in self.v_packets:
6                self.v_packets.remove(vpacket)
7              if packet_dict in self.v_packets:
8                self.v_packets.remove(packet_dict)
9              return True
10     else:
11         ... UDP/TCP ...
```

Again, the action would be to delete the two violation packets from the marked packet list and return a boolean value to signify what has occurred, which will make the function caller signify **sdn_dump.py** about the violation.

Listing 5.35: Forward Violation Found

```
1  ## Policy Violation Found: PACKET FORWARDING ##
2  > Packet: Sun 07:25:13 DevID: 00-00-00-00-00-01 ICMP:
       pkt ff:ff:ff:ff:ff:ff > ff:ff:ff:ff:ff:ff, ip
       10.0.0.1 > 10.0.0.2: 8:ECHO_REQUEST
3  > Packet: Sun 07:25:13 DevID: 00-00-00-00-00-02 ICMP:
       pkt ff:ff:ff:ff:ff:ff > ff:ff:ff:ff:ff:ff, ip
       10.0.0.1 > 10.0.0.2: 8:ECHO_REQUEST
4    violates:
5  > Rule: Date MON to SUN block host1 to host2
```

Matching a host data transmission against rules

sdn_dump.py also has the ability to check device port data against a network policy. It does this by forwarding its stored and sorted port data to the **VC's check_if_ports_legal()** function. The sorted data comes in form of dictionaries where a host IP is linked to a byte amount, there are four such dictionaries byte sent (*bsent*) and received (*brecv*), and the same for packet

sent (*psent*) and received (*precv*).

We have two possible rules for the Data-rule, one that says how much we can send and one that says how much we can receive. They are handled very similarly so we will show the one with receive restriction for convenience.

Listing 5.36: Data-rule examples

```
1 Data 1000 b from host2     # How much host 2 can send
2 Data 10 KB to 10.0.0.100   # How much 10.0.0.100 can
    recv
```

check_if_ports_legal() works by looping through all the rules that are of the Data-type and firstly extracting the IP address from the rule.

Listing 5.37: Step 1: check_if_ports_legal()

```
1  def check_if_ports_legal(self, bsent, brecv, psent,
       precv):
2      rule_ip = None
3      for rule in self.policy_rules_values:
4        if "Data" in rule["rule_type"]:
5          if "to" in rule: # what the IP  can recv
6            match = re.match(r'\d{1,3}\.\d{1,3}\.\d
                 {1,3}\.\d{1,3}', rule.get("to"))
7            if match:
8              rule_ip = rule.get("to")
9            else:
10             rule_ip = self.policy_hosts_name.get(rule.
                 get("to"))
```

From the code above this flow on **(Figure 5.5)** follows.

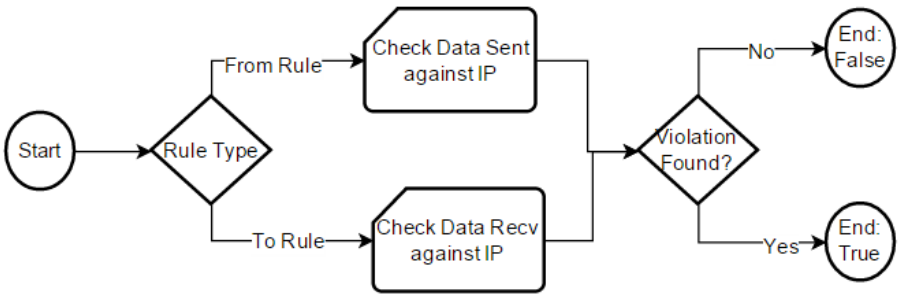

Figure 5.5: Flow for checking data transmission

Once we have an IP from the rule we check if it matches any of the IPs in the stored data in how much bytes has been received by each IP. If we find one, we extract the limit set by the rule and convert that into bytes.

Listing 5.38: Step 2: check_if_ports_legal()

```
# We now know the rule IP, lets match it agaisnt
    actual port stats
          for recv_stat in brecv:
            if recv_stat == rule_ip:
              rule_bytes = self.
                 convert_notation_to_bytes(rule.get("
                 lim"), rule.get("notation"))
              if int(rule_bytes) < int(brecv[recv_stat
                 ]):
                 ... print output ...
                 return True
        elif "from" in rule: # what the IP can send
           ... handle sent data restrictions ...
```

We then look at how much bytes the host can receive *(rule_bytes)* and how much the host has received *(brecv[recv_stat])*. If the host has received more than allowed, print an output on it and return a *True* boolean.

Converting rule limitation into bytes

The function **convert_notation_to_bytes()** will convert the rule limitation into bytes. The rule can have many different size notations (B, KB, MB etc.) and we need to be able to understand them all.

Listing 5.39: convert_notation_to_bytes()

```
def convert_notation_to_bytes(self, lim, notation):
    notation = notation.upper()
    if notation == "B":
        return int(lim)
    elif notation == "KB":
        return int(lim) * 1024
    elif notation == "MB":
        return int(lim) * 1024 * 1024
    elif notation == "GB":
        return int(lim) * 1024 * 1024 * 1024
    elif notation == "TB":
        return int(lim) * 1024 * 1024 * 1024 * 1024
    else:
        return 0
```

We see that for each increment in the size notation we multiply with 1024 more bytes. In the end if a violation is found at this level, we print out such an output and notify **sdn_dump.py**.

Listing 5.40: Data Violation Found

```
## Policy Violation Found: RECEIVED TO MUCH DATA ##
> IP: 10.0.0.1 address received: 1230 bytes
> IP: 10.0.0.1 can only receive: 1024 bytes
  violates:
```

```
5 > Rule: Data 1 KB to host1
```

5.2.6 Question A - Part 4: Automate Test Packets

The last part of Question A: automate network behaviour for the user. Our **automator.py** class will handle this section by getting the parsed rules from **violation_checker.py** and then creating and sending packets that would violate such rules.

Starting automation

We start the **automator.py** giving the **__init()__** function the parsed rules from **violation_checker.py**, the rule strings themselves, the links between host and IP address, and lastly how many switches our network contains.

Listing 5.41: Automator.py __init()__

```
1 def __init__(self, rules, rules_values, host_ips,
      host_names, switch):
2         self.rules = rules
3         self.rules_values = rules_values
4         self.host_ips = host_ips
5         self.host_names = host_names
6         self.switch_count = 0
7         self.switch_limit = int(switch)
8         core.openflow.addListenerByName("ConnectionUp"
              , self._handle_ConnectionUp)
```

We also set a switch count to 0 and assign a listener to *ConnectionUp* OpenFlow events. We want to wait with packet creation untill the last switch has turned on the network, before starting the automation of packet creation. We solve this by counting is *ConnectionUp* event until we reach a the switch limit, at which point we start the **type_decider()** function which will create a packet for each rule.

Listing 5.42: _handle_ConnectionUp()

```
1 def _handle_ConnectionUp(self,event):
2         self.switch_count += 1
3         if self.switch_count == self.switch_limit:
4             self.type_decider(event)
```

Decision maker

The **type_decider()** function will initiate creation and sending of test packets. It is done by looping through all the rules and looking at what protocol the rule encompass. E.g. an ICMP packet creation and sending is done by this code snippet:

Listing 5.43: type_decider() ICMP creation

```
1 def type_decider(self,event):
2     for rule in self.rules_values:
3         if "ICMP" in rule["prot"]:
4             src =  self.hostname_to_ip_for_from(rule)
5             dst =  self.hostname_to_ip_for_to(rule)
6             ping = self.create_ping(src, dst)
7             self.send_packets(event, ping)
8         elif "UDP" in rule["prot"]:
9             ...
```

If a rule can be tested by using ICMP packets, we extract the source and destination address. We then create a ping payload by using the function **create_ping()** and lastly send the packet using **send_packets()** function. Creating UDP and TCP packet very similar, but we also need to extract the source and destination port from the rules, and use their respective payload creating functions **create_udp()** and **create_tcp()**.

Create ICMP

Creating the ICMP is handled by **create_ping()** function. We first create the ICMP payload and then we need to add it to the IP payload, finally we return the packet to the caller function for sending.

Listing 5.44: create_ping()

```
1  def create_ping(self, src, dst):
2          # Make a ping request:
3          icmp = pkt.icmp()
4          icmp.type = pkt.TYPE_ECHO_REQUEST
5          echo = pkt.ICMP.echo(payload = "0123456789")
6          icmp.payload = echo
7
8          #Create IP packet
9          ip = pkt.ipv4()
10         ip.protocol = ip.ICMP_PROTOCOL
11         ip.srcip = IPAddr(src)
12         ip.dstip = IPAddr(dst)
13         ip.payload = icmp
14         return ip
```

Note that we need to use *pkt.TYPE_ECHO_REQUEST* as ICMP type for the packet, and we need to specify the IP protocol on the IP packet as *ip.ICMP_PROTOCOL*. The ICMP type and IP protocol specification and no port numbers are the major differences between this function at the two other creation functions (create UDP or TCP).

Create UDP

Creating UDP packets works very similar to creating ICMP packets. We first create the UDP payload, add it to the IP payload and return this to the caller function.

Listing 5.45: create_udp()

```
1  def create_udp(self, src, dst, sport, dport):
2          # Create UDP packet:
3          udp = pkt.udp()
4          udp.srcport = int(sport)
5          udp.dstport = int(dport)
6
7          # Create the IP:
8          ip = pkt.ipv4()
9          ip.protocol = ip.UDP_PROTOCOL
10         ip.srcip = IPAddr(src)
11         ip.dstip = IPAddr(dst)
12         ip.payload = udp
13         return ip
```

Note that in **line 4 and 5** set source and destination port in the UDP payload and then set UDP as the IP protocol for the IP packet.

Create TCP

TCP packets have a bit more to them, we need to set both port and IP, but also *SYN* flag, *SEQ* and *ACK* numbers, *window size* and *data offset* number.

Listing 5.46: create_tcp()

```
1  def create_tcp(self, src, dst, sport, dport):
2          # Create TCP:
3          tcp = pkt.tcp()
4          tcp.srcport = int(sport)
5          tcp.dstport = int(dport)
6          tcp._setflag(tcp.SYN_flag,1)
7          tcp.seq = 0
8          tcp.ack = 0
9          tcp.win = 1
10         tcp.off = 5
11
12         # Create the IP:
13         ip = pkt.ipv4()
14         ip.protocol = ip.TCP_PROTOCOL
15         ip.srcip = IPAddr(src)
16         ip.dstip = IPAddr(dst)
17         ip.payload = tcp
18         return ip
```

Again note the IP protocol changes to TCP in the IP packet on line 14 in the listing.

Sending created packet

Finally after each one of the packet payloads have been created we need to send the packet. **send_packets()** function does this for us. Firstly we create ethernet payload that is marked as an IP packet. Then we add the IP payload we created (ICMP, TCP or UDP packet returned from creation functions).

Listing 5.47: create_tcp()

```
1  def send_packets(self, event, ip_packet):
2          #Create Ethernet Payload
3          eth = ethernet()
4          eth.src = EthAddr("ff:ff:ff:ff:ff:ff")
5          eth.dst = EthAddr("ff:ff:ff:ff:ff:ff")
6          eth.type = eth.IP_TYPE
7          eth.payload = ip_packet
8
9          msg = of.ofp_packet_out()
10         msg.data =  eth.pack()
11         msg.in_port = of.OFPP_NONE
12         msg.actions.append(of.ofp_action_output(port =
               of.OFPP_CONTROLLER))
13         event.connection.send(msg)
```

Note that we use a mac broadcasting since we don't know each host's mac address at this point and we need some way for the network to send the packets. A result from this is that **l2_leaning** controller will not create flow tables from these packets since they don't contain any usable mac to create flow table entries. We could look at this as minimizing intrusion and change on the network when we create automatic network traffic. After that we use OpenFlow to create a OpenFlow message. We add the ethernet payload as message data, and set *of.OFPP_NONE* as message inport (meaning shouldn't be associated with a physical port[13]) and we set message action to be *of.OFPP_CONTROLLER* (meaning send packet first to controller rather than an switch outport[13]). By sending first to controller we can ensure that other controller apps running on the network will also see the created packet we are sending (all applications listening to *PacketIn* events).

Flow chart for sending packets

Creating and sending packets follows the flow on **(Figure 5.6)**.

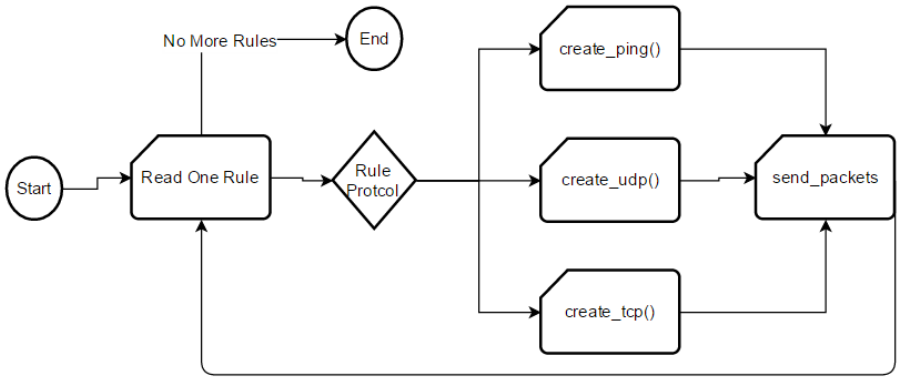

Figure 5.6: Flow for automating packet sending

5.3 Question B: Implementation

This section aims to explain the steps taken after a violation on Question A has been found. To recap, Question B is **"Does the device state match the policy?"**. This check needs to find if the network contains forwarding loops, black holes and disconnected switches. This implementation covers 2 out of 3, with a solution to black holes missing.

5.3.1 Question B: Initiation

Once **sdn_dump.py** gets notified that a violation has occurred on Question A it needs to set in motion a priority change of what to do next. We start by stopping the sending of packets to **violation_checker.py** since its job is done. We do this with the **sdn_dump.py** function called **decider()**.

Listing 5.48: Question B: sdn_dump.py decider()

```
1  def decider(bool_val, packet):
2    global layer1_correct
3    global last_packet
4
5    if bool_val:
6      last_packet = packet
7      print "### Stage 2: Started ###"
8      print "Question A: No, Policy does not match
           actual behaviour"
9      print "Searching Question B: 'Does policy match
           device state?'"
10     layer1_correct = False
11     con_check = check_switch_connectivity()
12     ...
```

This function is used on all the functions that request a violation reply from **violation_cheker**. We add the return value and the potentially violating packet to the function. And if we have a violation *(bool_val*

== *True)* then we set this packet as the last packet we received, and *layer1_correct* variable to *False*. By doing this we have told **sdn_dump** to stop sending more packet and transmission data to **violation_checker** for checking, and we take a general connectivity check of all the switches on the network by using the **check_switch_connectivity()**. This function will return us a boolean and give us an answer to Question B. The boolean will help us decide if we move to Question D (True/Yes) or C (False/No).

Listing 5.49: Question D: sdn_dump.py decider()

```
1  def decider(bool_val, packet):
2    ...
3    # Request flow stats (Layer 3 Question D)
4      if con_check: # if Yes on question B
5          print "### Stage 3: Started ###"
6          print "Question B: Yes, Device state matches
               Policy."
7          print "Searching  Question D: 'Does device
               state match hardware?'"
8          for con in core.openflow._connections.values()
               :
9              print "Requesting flow table entries from
                   device", con
10             con.send(of.ofp_stats_request(body=of.
                   ofp_flow_stats_request()))
```

If the *con_check* shows to be True (Yes, Device state matches Policy), we request the flow tables of each switch on the network so we can show the user what went wrong. We now wait for the **sdn_dump** to receive flow statistics from each network device, but before that we need to cover how the spanning tree was executed.

5.3.2 Question B: Spanning tree protocol

We also aim to solve or find issues that are frequently happening on the device state layer, for instance network forwarding loops that would influence the reachability on the network. As it happens *POX* has already implemented a nifty controller application that uses the spanning tree protocol, notifies and corrects every loop it finds on the network. We can use this application by importing it from the API.

Listing 5.50: Importing Spanning Tree

```
1  # spanning tree protocol from pox
2  import pox.openflow.discovery as discov
3  import pox.openflow.spanning_tree as spanning_tree
```

However we have to start the spanning tree at launch, because of its designed and implementation. The spanning tree listens to *ConnectionUp* events to graph the network links and then searches and corrects forward-

ing loops on the network. We use the function **start_spanning_tree()** to initiate the search.

Listing 5.51: Start Spanning Tree

```
1 def start_spanning_tree():
2     global yet_to_do
3     if yet_to_do:
4         print "Searching for forwarding loops with
              spanning tree..."
5         #pox.openflow.discovery
6         discov.launch()
7         #pox.openflow.spanning_tree --no-flood --hold-
              down
8         spanning_tree.launch()
9         yet_to_do = False
```

We only want this to be done once, so we keep the *yet_to_do* condition there for safety. We call this function at the **sdn_dump.py launch()** functions so that it can register the *ConnectionUp* events from the controller start.

5.4 Question D: Implementation

To answer Question D: "Does device state match the hardware?" we need to identify to the user what switch, rule and port sets[2] that are erroneous. We can do this by using the switch flow entries.

5.4.1 Find violating forwarding table entries

After we have found a violation on Question A, we need to provide the user what is wrong on the device state layer. One possibility is to give the operator an overview of the flow table entries that may have influenced the network behaviour in creating a policy violation. Essentially, this means to print out the table entries contributing the source and destination IPs that could have been used. We can print out the flow tables from the network devices once the flow stat request (we requested this from the **decider()**) has been received and return to our **handle_flow_requests()** listener function.

Print out flow table entries

Once we receive the flow stats from each network device we print out the related flow entries to our last packet. This way we show the operator what directly is influencing the wrongful network behaviour at the switch. We use the functions **handle_flow_request()** to print out each flow entries per device.

Listing 5.52: sdn_dump.py handle_flow_request()

```
1 def handle_flow_requests(event):
```

```
2       stats = flow_stats_to_list(event.stats)
3
4       ... Filter vars ...
5
6       print "## Flow Table for Deivce:", event.dpid, "##
            "
7       entry = 1
8       for stat in stats:
9           print "> Entry %d Match:" % entry
10          print stat["match"]
11          print ">> Entry Action"
12          print stat["actions"]
13          print ">> Byte Count"
14          print stat["byte_count"]
15          entry = entry + 1
```

Imagine we send a ping from 10.0.0.1 to 10.0.0.2 that creates a violation and we reach Question D after **l2_leaning** has created a flow entry for it, we should then find a related flow entire on the network device that show this violation.

Listing 5.53: Flow table output

```
1  ## Flow Table for Deivce: 1 ##
2  ... Other entries ...
3  > Entry 3 Match:
4  {'dl_type': 'IP', 'nw_dst': '10.0.0.2/32', 'dl_src': '
       e2:79:a7:67:cf:82', 'nw_proto': 1, 'nw_tos': 0, '
       tp_dst': 0, 'tp_src': 8, 'dl_dst': '86:04:a1:72:22:
       af', 'dl_vlan': 65535, 'nw_src': IPAddr('10.0.0.1')
       , 'in_port': 1}
5  >> Entry Action
6  [{'max_len': 0, 'type': 'OFPAT_OUTPUT', 'port': 2}]
7  >> Byte Count
8  98
```

We have now shown how the prototype finds what packet, device, *in_port* and *out_port* has created a violation using automation and the layered structure of SDN.

Chapter 6

Results and Analysis

6.1 Capturing Network Behaviour

sdn_dump has the ability to capture network behaviour across the network. It does this in all three operation modes. For the sake of this thesis, imagine a network with 2 switches that we run the controller on and create some manual network traffic.

6.1.1 Hosts, Devices and IPs

The test networks we use will have the standard Mininet hostnames (h1,h2, etc.), IP-addresses for these hostnames will also be (10.0.0.1, 10.0.0.2, etc.) and the device identification will be (00-00-00-00-00-01, 00-00-00-00-00-02, etc.).

6.1.2 Capturing ICMP

In this test we are sending an ICMP packet from **h1** (10.0.0.1) to **h2** (10.0.0.2) over a 2 switch network. We are monitoring the network behaviour using **sdn_dump** in operation **mode=1**.

Listing 6.1: Monitor ICMP traffic

```
1 Sun 14:50:49 DevID: 00-00-00-00-00-01 ICMP: pkt
    32:15:54:e1:e9:81 > 9e:e9:4a:cb:c4:4f, ip 10.0.0.1
    > 10.0.0.2: 8:ECHO_REQUEST
2 Sun 14:50:49 DevID: 00-00-00-00-00-02 ICMP: pkt
    32:15:54:e1:e9:81 > 9e:e9:4a:cb:c4:4f, ip 10.0.0.1
    > 10.0.0.2: 8:ECHO_REQUEST
3 Sun 14:50:49 DevID: 00-00-00-00-00-02 ICMP: pkt 9e:e9
    :4a:cb:c4:4f > 32:15:54:e1:e9:81, ip 10.0.0.2 >
    10.0.0.1: 0:ECHO_REPY
4 Sun 14:50:49 DevID: 00-00-00-00-00-01 ICMP: pkt 9e:e9
    :4a:cb:c4:4f > 32:15:54:e1:e9:81, ip 10.0.0.2 >
    10.0.0.1: 0:ECHO_REPY
```

With the monitor we can see that an ICMP request from **h1** arrived at device 1, then forwarded to device 2, which then delivered it to **h2** on **Sunday 14:50:49**. Using the monitor output we then see an ICMP reply coming from **h2**, back the same network path device 2 -> device 1, and arriving at **h1**. With the monitor we both see the date, packet path, packet type, packet source and destination.

6.1.3 Capturing TCP

In this example we manually create TCP traffic that we can monitor using the *iperf* command at the mininet terminal.

Listing 6.2: Monitor TCP traffic

```
1 Sun 14:50:56 DevID: 00-00-00-00-00-01 TCP: pkt
    32:15:54:e1:e9:81 > 9e:e9:4a:cb:c4:4f, ip 10.0.0.1
    > 10.0.0.2: srcp 49432 dstp 5001, seq 2611028079,
    ack 0, flags 2
2 Sun 14:50:56 DevID: 00-00-00-00-00-02 TCP: pkt
    32:15:54:e1:e9:81 > 9e:e9:4a:cb:c4:4f, ip 10.0.0.1
    > 10.0.0.2: srcp 49432 dstp 5001, seq 2611028079,
    ack 0, flags 2
3 Sun 14:50:56 DevID: 00-00-00-00-00-02 TCP: pkt 9e:e9:4
    a:cb:c4:4f > 32:15:54:e1:e9:81, ip 10.0.0.2 >
    10.0.0.1: srcp 5001 dstp 49432, seq 3538410437, ack
     2611028080, flags 18
4 Sun 14:50:56 DevID: 00-00-00-00-00-01 TCP: pkt 9e:e9:4
    a:cb:c4:4f > 32:15:54:e1:e9:81, ip 10.0.0.2 >
    10.0.0.1: srcp 5001 dstp 49432, seq 3538410437, ack
     2611028080, flags 18
```

As shown in the ICMP example shown above, using the **sdn_dump** monitor we can see the network path a packet takes across the network. Here we can observe that the same is true for the TCP packet with the addition of showing protocol flags and ports.

6.1.4 Other and UNKNOWN packet types

The monitor also captures ARP and UDP packets, and will work the same way as the two examples above. If the monitor captures a packet with another protocol, it will print and mark it as UNKNOWN. These packets are assumed to be OpenFlow related traffic.

6.1.5 Scalability of capturing

The monitoring mode was also tested on bigger networks. The transcript file **"mode1-10switch-ping.txt"** in the appendix shows the tool monitoring network wide traffic, while pinging from **h1** to **h10**. Here you can see the ping request and reply jump through 10 switches live.

6.2 Finding Policy Violation

Using **sdn_dump** in operation mode 2 or 3 will command it also to check the captured packets for policy violations. When we find a policy violation, we will in quick succession check Question A, B and D. The result of each of these Questions will be covered here. To capture a policy violation we have to write an example policy we can test.

6.2.1 Question A: Finding Time violation

Let us say we want to block the h1 to h2 connection between 10:00 - 23:00. Then the policy would read like this:

Listing 6.3: Time policy example

```
1 host1 = 10.0.0.1
2 host2 = 10.0.0.1
3
4 Time 10:00 to 23:00 block host1 to host2
```

This should mark ICMP, TCP and UDP communication between the two as a policy violation. We expect the tool to warn the user if such behaviour has occurred. The output would be like this:

Listing 6.4: Time Violation - file: ping-time-violation-found.txt

```
1 ... transcript shortened ...
2 Mon 15:40:39 DevID: 00-00-00-00-00-01 ICMP: pkt e2:79:
    a7:67:cf:82 > 86:04:a1:72:22:af, ip 10.0.0.1 >
    10.0.0.2: 8:ECHO_REQUEST
3 Mon 15:40:39 DevID: 00-00-00-00-00-02 ICMP: pkt e2:79:
    a7:67:cf:82 > 86:04:a1:72:22:af, ip 10.0.0.1 >
    10.0.0.2: 8:ECHO_REQUEST
4 ## Policy Violation Found: PACKET FORWARDING ##
5 > Packet: Mon 15:40:39 DevID: 00-00-00-00-00-01 ICMP:
    pkt e2:79:a7:67:cf:82 > 86:04:a1:72:22:af, ip
    10.0.0.1 > 10.0.0.2: 8:ECHO_REQUEST
6 > Packet: Mon 15:40:39 DevID: 00-00-00-00-00-02 ICMP:
    pkt e2:79:a7:67:cf:82 > 86:04:a1:72:22:af, ip
    10.0.0.1 > 10.0.0.2: 8:ECHO_REQUEST
7   violates:
8 > Rule: Time 10:00 to 23:00 block host1 to host2
```

We see that the application has seen an illegal packet being forwarded across the network, and warns the user by specifying what two packets and rule are being violated, providing us with the policy violation. We also observe from the two packets, what device(s) the violation occurred on. After this the tool will initiate a Question B search on the network. We will cover this in a later subsection.

6.2.2 Question A: Finding Date violation

Say, we would like to block h1 to h2 connection all week. Then the policy would read like this:

Listing 6.5: Time policy example

```
1 host1 = 10.0.0.1
2 host2 = 10.0.0.1
3
4 Date MON to SUN block host1 to host2
```

This should mark ICMP, TCP and UDP packets in that week interval as violating packets. We expect the tool to warn the user if such behaviour has occurred. Here is a result of trying to send a TCP packet between the two on a Sunday.

Listing 6.6: Date Violation - file: tcp-date-violation-found.txt

```
1 ... transcript shortened ...
2 Sun 16:53:44 DevID: 00-00-00-00-00-01 TCP: pkt aa:46:
    cb:d3:9d:c9 > c6:80:16:13:f2:79, ip 10.0.0.1 >
    10.0.0.2: srcp 58421 dstp 5001, seq 697520508, ack
    0, flags 2
3 Sun 16:53:44 DevID: 00-00-00-00-00-02 TCP: pkt aa:46:
    cb:d3:9d:c9 > c6:80:16:13:f2:79, ip 10.0.0.1 >
    10.0.0.2: srcp 58421 dstp 5001, seq 697520508, ack
    0, flags 2
4 ## Policy Violation Found: PACKET FORWARDING ##
5 > Packet: Sun 16:53:44 DevID: 00-00-00-00-00-01 TCP:
    pkt aa:46:cb:d3:9d:c9 > c6:80:16:13:f2:79, ip
    10.0.0.1 > 10.0.0.2: srcp 58421 dstp 5001, seq
    697520508, ack 0, flags 2
6 > Packet: Sun 16:53:44 DevID: 00-00-00-00-00-02 TCP:
    pkt aa:46:cb:d3:9d:c9 > c6:80:16:13:f2:79, ip
    10.0.0.1 > 10.0.0.2: srcp 58421 dstp 5001, seq
    697520508, ack 0, flags 2
7   violates:
8 > Rule: Date MON to SUN block host1 to host2
```

Again the application sees violating behaviour, it warns the user and points out what is wrong with the behaviour. As mentioned earlier, the application would initiate a search on Question B.

6.2.3 Question A: Finding Data violation

In order for us to limit the number of bytes host2 can send. Then the policy would read like this:

Listing 6.7: Time policy example

```
1 host1 = 10.0.0.1
```

```
2 host2 = 10.0.0.1
3
4 Data 1000 b from host2
```

We expect the application to find a violation if **host2** transmits more data than allowed.

Listing 6.8: Data Violation - file: ping-data-violation-found.txt

```
1 ... transcript shortened ...
2 Sun 16:56:17 DevID: 00-00-00-00-00-01 ICMP: pkt c6
    :80:16:13:f2:79 > aa:46:cb:d3:9d:c9, ip 10.0.0.2 >
    10.0.0.1: 0:ECHO_REPY
3 Sun 16:56:18 DevID: 00-00-00-00-00-01 ICMP: pkt aa:46:
    cb:d3:9d:c9 > c6:80:16:13:f2:79, ip 10.0.0.1 >
    10.0.0.2: 8:ECHO_REQUEST
4 ## Policy Violation Found: SENDING TO MUCH DATA ##
5 > IP: 10.0.0.2 address sent: 21503948 bytes
6 > IP: 10.0.0.2 can only send: 1000 bytes
7   violates:
8 > Rule: Data 1000 b from host2
```

We see that after a few packets have been sent, the application will request port data from the devices and check if **host2** has transmitted more data than allowed.

6.2.4 Question B: Checks

With Question B we are looking for loops and disconnected devices.

Checking for disconnected network devices

Following the time violation found from the transcript of **"ping-time-violation-found.txt"**, we get the transcript on Listing 6.9.

Listing 6.9: Time Violation: Step 2 - file: ping-time-violation-found.txt

```
1 ... transcript shortened ...
2 ### Stage 2: Started ###
3 Question A: No, Policy does not match actual behaviour
4 Searching Question B: 'Does policy match device state
    ?'
5 Checking switch connectivity:
6 Switch [00-00-00-00-00-01 2] is alive
7 Switch [00-00-00-00-00-02 1] is alive
8 0 device(s) unaccounted for
9 ### Stage 3: Started ###
10 Question B: Yes, Device state matches Policy.
11 Searching  Question D: 'Does device state match
    hardware?'
```

As a result, we see that two devices are alive as expected and 0 are unaccounted for. The application will take this as a *Yes* on Question B, and move down to Question D. If we told the application to expect more devices than what OpneFlow registered as connected, the result would be different:

Listing 6.10: Disconnected Device Found - file: ping-data-violation-found.txt

```
1  ... transcript shortened ...
2  ### Stage 2: Started ###
3  Question A: No, Policy does not match actual behaviour
4  Searching Question B: 'Does policy match device state
       ?'
5  Checking switch connectivity:
6  Switch [00-00-00-00-00-01 2] is alive
7  Switch [00-00-00-00-00-02 1] is alive
8  1 device(s) unaccounted for
9  ### Stage 3: Started
10 Question B: No, Device state doesn't match Policy.
11 Searching  Question C: 'Does physical view  match
       Device State?'
```

The application sees a device is missing and moves to Question C rather then Question D.

Checking for forwarding loops

Finding and solving forwarding loops is solved by using POX's own spanning tree module. The resulting output shows the module calculating the network links, and disabling the ones that would create a loop.

Listing 6.11: Spanning tree - file: ping-data-violation-found.txt

```
1  ... transcript shortened ...
2  INFO:openflow.discovery:link detected:
       00-00-00-00-00-02.2 -> 00-00-00-00-00-01.2
3  INFO:openflow.discovery:link detected:
       00-00-00-00-00-01.2 -> 00-00-00-00-00-02.2
4  INFO:openflow.spanning_tree:4 ports changed
```

This part hasn't been tested so extensively since it is already part of the POX API. The modules ability to solve network loops (disable links that creating loops), it's need to be started from the **launch()** and not from the place we start the Question B search, and no ability to extract a *yes* and *no* answer from the module, its result isn't taken into account when deciding what question to search next in the decision tree.

6.2.5 Question D: Violating flow entries, devices and port sets

Question D is one of the leafs of the decision tree. At the end of it's search the user should also learn what flow entries, *inport* and *outport* is creating

violating behaviour on the network.

Expected result

As stated above, the expected result from this is for the user to learn what flow entry, inport and outport is creating violating behaviour. If we look at **"ping-data-violation-found.txt"** extract we can see what an expected result would look like. At the network device 1, we see two flows, one for each direction.

Listing 6.12: Spanning tree - file: ping-data-violation-found.txt

```
1  ... transcript shortened ...
2  ## Flow Table for Deivce: 1 ##
3  ... transcript shortened ...
4  > Entry 2 Match:
5  {'dl_type': 'IP', 'nw_dst': '10.0.0.1/32', 'dl_src': '
       c6:80:16:13:f2:79', 'nw_proto': 1, 'nw_tos': 0, '
       tp_dst': 0, 'tp_src': 0, 'dl_dst': 'aa:46:cb:d3:9d:
       c9', 'dl_vlan': 65535, 'nw_src': IPAddr('10.0.0.2')
       , 'in_port': 2}
6  >> Entry Action
7  [{'max_len': 0, 'type': 'OFPAT_OUTPUT', 'port': 1}]
8  >> Byte Count
9  196
10 > Entry 3 Match:
11 {'dl_type': 'IP', 'nw_dst': '10.0.0.2/32', 'dl_src': '
       aa:46:cb:d3:9d:c9', 'nw_proto': 1, 'nw_tos': 0, '
       tp_dst': 0, 'tp_src': 8, 'dl_dst': 'c6:80:16:13:f2
       :79', 'dl_vlan': 65535, 'nw_src': IPAddr
       ('10.0.0.1'), 'in_port': 1}
12 >> Entry Action
13 [{'max_len': 0, 'type': 'OFPAT_OUTPUT', 'port': 2}]
14 >> Byte Count
15 98
```

However because of the timing issue between which controller application's **(l2_learning or sdn_dump)** process will finish first, we can get unexpected results.

Unexpected results

Once an event such as *packetIn* arrives to the listener, it will be processed by the handler linked with that listener and event. If we have two listeners to the same event, such as in this case **sdn_dump** and **l2_learning**, it becomes an issue of timing. Does **sdn_dump** find a violation, request flow stats, process the flow stats and print them out, before **l2_learning** gets the same packet (same event that triggered a violation at **sdn_dump**), calculates a flow entry and sends it to the same device **sdn_dump** is requesting flow stats from? If this is the case, then **sdn_dump** is processing empty

flow stats or flow stats without the violating entry in them. It is still being created at **l2_learning**. This timing issue resulted in irregular results on Question D.

Irregular results such as these, in this **sdn_dump** request found a violation and requested flow stats before **l2_learning** had time to add any flow entries. The only thing present is the devices connection to the controller.

Listing 6.13: Only Control Flow - file: tcp-date-violation-found.txt

```
1  ... transcript shortened ...
2  ## Flow Table for Deivce: 2 ##
3  > Entry 1 Match:
4  {'dl_type': 'LLDP', 'dl_dst': '01:23:20:00:00:01'}
5  >> Entry Action
6  [{'max_len': 65535, 'type': 'OFPAT_OUTPUT', 'port': '
       OFPP_CONTROLLER'}]
7  >> Byte Count
8  0
9  ## Flow Table for Deivce: 1 ##
10 > Entry 1 Match:
11 {'dl_type': 'LLDP', 'dl_dst': '01:23:20:00:00:01'}
12 >> Entry Action
13 [{'max_len': 65535, 'type': 'OFPAT_OUTPUT', 'port': '
       OFPP_CONTROLLER'}]
14 >> Byte Count
15 41
```

Most of the times there is a mix between the two, where you see some flow entries on device or incomplete pairs (flow entry for the request and reply). But an incomplete pair doesn't necessary mean the result is wrong. In the case of **"ping-date-violation-found.txt"** we have a case of forwarding violation. This means that we find a violation when it is forwarded to another device. Because we at this point don't expect to see a flow for the reply, we found the violation before it reached its designated host and no reply packet has been created.

Listing 6.14: No Pair Expected - file: ping-date-violation-found.txt

```
1  ... transcript shortened ...
2  ## Policy Violation Found: PACKET FORWARDING ##
3  > Packet: Sun 16:52:43 DevID: 00-00-00-00-00-01 ICMP:
       pkt aa:46:cb:d3:9d:c9 > c6:80:16:13:f2:79, ip
       10.0.0.1 > 10.0.0.2: 8:ECHO_REQUEST
4  > Packet: Sun 16:52:43 DevID: 00-00-00-00-00-02 ICMP:
       pkt aa:46:cb:d3:9d:c9 > c6:80:16:13:f2:79, ip
       10.0.0.1 > 10.0.0.2: 8:ECHO_REQUEST
5    violates:
6  > Rule: Date MON to SUN block host1 to host2
7  ... transcript shortened ...
```

```
## Flow Table for Deivce: 1 ##
... transcript shortened ...
> Entry 3 Match:
{'dl_type': 'IP', 'nw_dst': '10.0.0.2/32', 'dl_src': '
   aa:46:cb:d3:9d:c9', 'nw_proto': 1, 'nw_tos': 0, '
   tp_dst': 0, 'tp_src': 8, 'dl_dst': 'c6:80:16:13:f2
   :79', 'dl_vlan': 65535, 'nw_src': IPAddr
   ('10.0.0.1'), 'in_port': 1}
>> Entry Action
[{'max_len': 0, 'type': 'OFPAT_OUTPUT', 'port': 2}]
>> Byte Count
98
```

In the transcript above you can notice a violation flow entry (entry 3) on device 1, which is an expected result. However, on the second device **l2_learning** still hasn't processed and created a flow entry for the application to print out. This is what is creating the irregularities. The experiments also showed that packets created with the use of the **automator.py**, didn't create flows either. The suspected culprit for this was the use of "ff:ff:ff:ff:ff:ff" macs on the Ethernet packet payload, rather than timing, There was no evidence to indicate that **l2_learning** would create flow entries for mac broadcasting address after multiple tests.

6.3 Automate Network Behaviour

The tests shown above have the ability to be fully automated. This means that the user doesn't have to create test traffic manually, but let the tool itself test the policy rules. This done by running the application in the third operational mode.

6.3.1 Create ICMP packets

In this test we want the tool to create its own testing network behaviour through the use of policy rules. We have the policy:

Listing 6.15: Example Policy: Create ICMP Packets

```
host1 = 10.0.0.1
host2 = 10.0.0.2

Time 10:00 to 23:00 block host1 to host2
```

We expect to see the tool create an ICMP packet (since no protocol is specified in the policy) from **host1** to **host2** and send it on the network. We also expect to see a Packet Return violation message on Question A to appear. The resulting output from **sdn_dump** on a one switch network:

Listing 6.16: Create ICMP Packet - File: auto-ping-reply.txt

```
1  mininet@mininet-vm:~/pox$ ./pox.py pox.forwarding.
       l2_learning pox.sdntroubleshoot.sdn_dump --switch=1
       --mode=3
2  ... transcript shortened ...
3  Sun 17:07:12 DevID: 00-00-00-00-00-01 ICMP: pkt ff:ff:
       ff:ff:ff:ff > ff:ff:ff:ff:ff:ff, ip 10.0.0.1 >
       10.0.0.2: 8:ECHO_REQUEST
4  Sun 17:07:12 DevID: 00-00-00-00-00-01 ARP: pkt fe:67:
       e8:4c:2e:12 > ff:ff:ff:ff:ff:ff, hw fe:67:e8:4c:2e
       :12 > 00:00:00:00:00:00: 1:REQUEST
5  Sun 17:07:12 DevID: 00-00-00-00-00-01 ARP: pkt 4e:ce
       :20:54:1b:9a > fe:67:e8:4c:2e:12, hw 4e:ce:20:54:1b
       :9a > fe:67:e8:4c:2e:12: 2:REPLY
6  Sun 17:07:12 DevID: 00-00-00-00-00-01 ICMP: pkt fe:67:
       e8:4c:2e:12 > 4e:ce:20:54:1b:9a, ip 10.0.0.2 >
       10.0.0.1: 0:ECHO_REPY
7  ## Policy Violation Found: PACKET RETURN ##
8  > Packet: Sun 17:07:12 DevID: 00-00-00-00-00-01 ICMP:
       pkt ff:ff:ff:ff:ff:ff > ff:ff:ff:ff:ff:ff, ip
       10.0.0.1 > 10.0.0.2: 8:ECHO_REQUEST
9  > Packet: Sun 17:07:12 DevID: 00-00-00-00-00-01 ICMP:
       pkt fe:67:e8:4c:2e:12 > 4e:ce:20:54:1b:9a, ip
       10.0.0.2 > 10.0.0.1: 0:ECHO_REPY
10   violates:
11 > Rule: Time 10:00 to 23:00 block host1 to host2
```

We can see the packet ICMP request being sent to Device 1, with source IP 10.0.0.1 and destination IP 10.0.0.2. We can also see the distinct use of ff:ff:ff:ff:ff:ff mac on the packets since the policy only knows the IP of assigned hosts. Then we see **host2** using ARP to figure out mac addresses and lastly send an ICMP reply to **host1**. Since the policy states that this connection should be blocked at this time interval, we also see a Policy Violation Found message, and the application starts to troubleshoot downward the tree.

6.3.2 Create TCP packets

In this experiment we wanted the tool to test blocking a TCP connection, but this time around using the automatic creation of TCP packets. The policy example is:

Listing 6.17: Example Policy: Create ICMP Packets

```
1  host1 = 10.0.0.1
2  host2 = 10.0.0.2
3
4  Time 10:00 to 23:00 block host1 to host2 prot TCP
```

When we test this on a network with two switches, we expect to see a TCP packet being created from **host1** to **host2**. Another anticipation of this

experiment is a violation of Question A.

Listing 6.18: Create TCP Packet - File: auto-tcp-forward.txt

```
1 mininet@mininet-vm:~/pox$ ./pox.py pox.forwarding.
    l2_learning pox.sdntroubleshoot.sdn_dump --switch=2
    --mode=3
2 ... transcript shortened ...
3 Sun 17:08:40 DevID: 00-00-00-00-00-01 TCP: pkt ff:ff:
    ff:ff:ff:ff > ff:ff:ff:ff:ff:ff, ip 10.0.0.1 >
    10.0.0.2: srcp 48966 dstp 5001, seq 0, ack 0, flags
    2
4 Sun 17:08:40 DevID: 00-00-00-00-00-02 TCP: pkt ff:ff:
    ff:ff:ff:ff > ff:ff:ff:ff:ff:ff, ip 10.0.0.1 >
    10.0.0.2: srcp 48966 dstp 5001, seq 0, ack 0, flags
    2
5 ## Policy Violation Found: PACKET FORWARDING ##
6 > Packet: Sun 17:08:40 DevID: 00-00-00-00-00-01 TCP:
    pkt ff:ff:ff:ff:ff:ff > ff:ff:ff:ff:ff:ff, ip
    10.0.0.1 > 10.0.0.2: srcp 48966 dstp 5001, seq 0,
    ack 0, flags 2
7 > Packet: Sun 17:08:40 DevID: 00-00-00-00-00-02 TCP:
    pkt ff:ff:ff:ff:ff:ff > ff:ff:ff:ff:ff:ff, ip
    10.0.0.1 > 10.0.0.2: srcp 48966 dstp 5001, seq 0,
    ack 0, flags 2
8   violates:
9 > Rule: Time 10:00 to 23:00 block host1 to host2 prot
    TCP
```

From the transcript we can observe the TCP packet being sent trough device 1 to device 2, and the **sdn_dump** detecting the transmission as violating behaviour.

Unwanted behaviour

The implementation of this experiment did however create some unwanted behaviour. During the automation operation mode, we observed network violations being created, not by the network, but by the tool itself. If we specified the policy to block a TCP connection both ways, such as this:

Listing 6.19: Policy that would create erroneous behaviour during automation

```
1 Time 10:00 to 23:00 block h1 to h2 prot TCP sport
    41238 dport 5001
2 Time 10:00 to 23:00 block h2 to h1 prot TCP sport 5001
    dport 41238
```

When the prototype then created packets to test both these rules and send them to the controller, the controller would mark them both as violating packets. And then view the two packets as violation network

behaviour by specify it as a violating packet return to the first packet. This only happens when a connection was specified with two policy rules, such as shown above.

6.3.3 Create UDP packets

In this experiment we use the tool to test a blocked UDP connection to and from a specific host, but using the automatic creation of UDP packets. The policy example is:

Listing 6.20: Example Policy: Create ICMP Packets

```
1 host1 = 10.0.0.1
2 host2 = 10.0.0.2
3
4 Time 10:00 to 23:00 block host2 prot UDP
```

The experiment will try to create violating behaviour by reading the rule and creating a UDP packet to test it. The expected result is that the **automator** will create a packet from host2 and send it to an imaginary IP address. Again we expect to see a violation to occur with Question A.

Listing 6.21: Create TCP Packet - File: auto-tcp-forward.txt

```
1 mininet@mininet-vm:~/pox$ ./pox.py pox.forwarding.
    l2_learning pox.sdntroubleshoot.sdn_dump --switch=2
    --mode=3
2 ... transcript shortened ...
3 Sun 17:11:26 DevID: 00-00-00-00-00-01 UDP: pkt ff:ff:
    ff:ff:ff:ff > ff:ff:ff:ff:ff:ff, ip 10.0.0.2 >
    10.0.1.100: srcp 48966 dstp 5001
4 Sun 17:11:26 DevID: 00-00-00-00-00-02 UDP: pkt ff:ff:
    ff:ff:ff:ff > ff:ff:ff:ff:ff:ff, ip 10.0.0.2 >
    10.0.1.100: srcp 48966 dstp 5001
5 ## Policy Violation Found: PACKET FORWARDING ##
6 > Packet: Sun 17:11:26 DevID: 00-00-00-00-00-01 UDP:
    pkt ff:ff:ff:ff:ff:ff > ff:ff:ff:ff:ff:ff, ip
    10.0.0.2 > 10.0.1.100: srcp 48966 dstp 5001
7 > Packet: Sun 17:11:26 DevID: 00-00-00-00-00-02 UDP:
    pkt ff:ff:ff:ff:ff:ff > ff:ff:ff:ff:ff:ff, ip
    10.0.0.2 > 10.0.1.100: srcp 48966 dstp 5001
8   violates:
9 > Rule: Time 10:00 to 23:00 block host2 prot UDP
```

Again as expected, we see the UDP packet being created, sent across the network and **sdn_dump** detecting it as a violation.

6.4 Case: Firewall Policy

How does the tool work with behaviour altering controller applications? The experiments here try to answer this question. We don't expect to see

violations occurring when **mac_blocker** or **blocker** is correctly blocking TCP traffic as specified in a policy.

6.4.1 Run sdn_dump with a firewall

The POX **mac_blocker** or **blocker**, is a tool that comes with POX. It blocks specific TCP ports which are given as parameters. Then experiment will then run **sdn_dump** in operation mode 3, where it will try to create a TCP packet with *destination port 5001*. Based on the policy this should be blocked and set as a violation if allowed to be forwarded across the network.

Listing 6.22: Policy during firewall test

```
1 host1 = 10.0.0.1
2 host2 = 10.0.0.2
3
4 Time 10:00 to 23:00 block host1 to host2 prot TCP
    sport 41238 dport 5001
```

We will run this on a two host and two switch network and see if the packet will create a forwarding violation without the firewall and no violation with the firewall.

Run without blocker.py

If we firstly run the controller application without the firewall **blocker.py**, we expect to see a violation to occur based on the policy we have written.

Listing 6.23: Run sdn_dump.py without a firewall - file: without-firewall.txt

```
1 mininet@mininet-vm:~/pox$ ./pox.py pox.forwarding.
    l2_learning pox.sdntroubleshoot.sdn_dump --switch=2
     --mode=3
2 ... transcript shortened ...
3 Sun 17:43:13 DevID: 00-00-00-00-00-01 TCP: pkt ff:ff:
    ff:ff:ff:ff > ff:ff:ff:ff:ff:ff, ip 10.0.0.1 >
    10.0.0.2: srcp 41238 dstp 5001, seq 0, ack 0, flags
     2
4 Sun 17:43:13 DevID: 00-00-00-00-00-02 TCP: pkt ff:ff:
    ff:ff:ff:ff > ff:ff:ff:ff:ff:ff, ip 10.0.0.1 >
    10.0.0.2: srcp 41238 dstp 5001, seq 0, ack 0, flags
     2
5 ## Policy Violation Found: PACKET FORWARDING ##
6 > Packet: Sun 17:43:13 DevID: 00-00-00-00-00-01 TCP:
    pkt ff:ff:ff:ff:ff:ff > ff:ff:ff:ff:ff:ff, ip
    10.0.0.1 > 10.0.0.2: srcp 41238 dstp 5001, seq 0,
    ack 0, flags 2
7 > Packet: Sun 17:43:13 DevID: 00-00-00-00-00-02 TCP:
    pkt ff:ff:ff:ff:ff:ff > ff:ff:ff:ff:ff:ff, ip
```

```
    10.0.0.1 > 10.0.0.2: srcp 41238 dstp 5001, seq 0,
    ack 0, flags 2
8   violates:
9 > Rule: Time 10:00 to 23:00 block host1 to host2 prot
    TCP sport 41238 dport 5001
```

As expected without running the firewall in order to stop the created packet from being forwarded, **sdn_dump** will warn the user of a violation that has occurred.

Run with blocker.py

When we run the same experiment with the blocker we expect **sdn_dump** to create a TCP packet, send it to a device, but not see any violation occur. The blocker will ensure that the packet is dropped, and thereby the policy hasn't been violated.

Listing 6.24: Run sdn_dump with a firewall - file: with-firewall.txt

```
1 mininet@mininet-vm:~/pox$ ./pox.py pox.forwarding.
    l2_learning pox.sdntroubleshoot.sdn_dump --switch=2
    --mode=3 pox.forwarding.blocker --ports=5001
2 ... transcript shortened ...
3 Sun 17:44:16 DevID: 00-00-00-00-00-01 TCP: pkt ff:ff:
    ff:ff:ff:ff > ff:ff:ff:ff:ff:ff, ip 10.0.0.1 >
    10.0.0.2: srcp 41238 dstp 5001, seq 0, ack 0, flags
    2
```

With this we know that **sdn_dump** does not find violations if the policy is upheld by the actual network behaviour.

6.5 Overall Analysis

Overall the experiments show that the prototype does as expected. It monitors network behaviour, it finds and points out violation behaviour using the policy description, it has the ability to automate testing of policy rules, and it shows the user what switch flows and ports are part of the network problem. However, we also found unexpected results in Question D. These results pointed out that the timing and the time taken to process events for different controller applications was the most obvious culprit.

With the issue of timing and irregular results, we could note the following: The irregularity doesn't necessarily mean that the solution is wrong, we need a way to delay the request of flow stats after **l2_learning** has finished processing the *packetIn* event and created a flow entry to the device. The issue of no flow tables registered was also seen when using operation mode 3. The packets were sent with "ff:ff:ff:ff:ff:ff" mac, since we don't assign mac addresses in the policy. It was unclear if we don't see flows from these packets because of the same timing issue we see with normal packets, or

that no flows are created for the broadcasting mac address by **l2_learning**.
The latter does however seem more likely.

Chapter 7

Discussion

The following chapter will address the purpose and choices taken during the approach, design and implementation phase of the thesis. Simultaneously, it will discuss the results derived from this approach, including an attempt to gauge the challenges faced and their impact. A final conclusion will be delivered in the next chapter.

7.1 Evaluating the Prototype

We set out to automate Heller's model for using the layered structure of SDN to troubleshoot a network. Heller delivered a decision tree which we could systematically follow, and there by automate.

7.1.1 Question A

Question A: Does the actual network behaviour match the policy? The goal was to design and implement a solution to this answer with automation. By the end of Question A, we expected to find what behaviour (packet and packet path) and rule were violating our policy. We specified that three parts are needed for this to be fully automated; detection of network behaviour, checking actual behaviour against intended and finally create testing network behaviour. We showed with the prototype and results that this could be implemented. We showed that we could describe intended network behaviour, and both check traffic we created manually ourselves, as well as allowed the tool itself to create testing traffic through automated packets from the rule description. However, we do need to note some limitations with the prototype as a result of our experiments. The developed descriptive policy language covered a small sample of the policy domain. We covered Time, Date and Data specific rules, but there are so many more thing we can cover under what constitutes a network policy.

We also saw a bug here while using operation mode 3. If we described a connection with two policy rules, one for the request and one for the reply, it would create packets for both and send them. The controller would then see this as a violation created by the network rather than the proto-

type itself. This could be solved in numerous ways, but the easiest solution would be for the prototype to see during parsing that both rules are covering the same connection, and that only one test packet is needed for the two rules.

7.1.2 Question B

Question B: Does the device state match policy? Here we needed to find common reachability problems such as forwarding loops, black holes and disconnectivity issues. The implementation handled loops using the spanning tree provided by POX, and found disconnected devices by counting the connections OpenFlow kept of the connected network devices. We could have solved forwarding loops problem even better if there was a way for us to start the spanning tree once we are troubleshooting at Question B, rather then at launch. And have an ability to extract a *yes* or *no* answer from it when it was finished in order to use this information in the decision making process after Question B. We could also have expanded the idea of disconnectivity issues by having a better description in the policy, i.e. what is connected to what port on what device. This would allow for an expanded ability to check if both devices, hosts and all ports are correctly connected.

The biggest limitation on implementation of this question, was that a solution to black holes wasn't researched. In the beginning the goal was to use capabilities of Hassel[26] to solve all three of these issues, but the technical difficulties and time allocated to research a way to incorporate the tool Hassel turned the approach in another direction.

7.1.3 Question D

Question D: Does the device state match hardware? By the end of this equivalence check the user should know what behaviour, policy rule, devices, device *inport* and *outport* are creating a violation. The implementation showed that this could be done, but that the result was unreliable because of timing. The initial idea was to print and filter out packets based on the last violating packet received, but that design proved itself as challenging.

A flow entry is created and set on a switch by a **l2_learning** or **l3_learning** controller application delivered with POX. Once a *packetIn* event is sent to the controller, each controller application listening to that event will process and handle it. So while **sdn_dump** and **violation_checker** are reading and checking a packet for violation, the **l2_learning** application will process the same packet and create a related flow entry for that specific device which created the event. This causes a timing issues. Depending on which process will finished first, often **sdn_dump** would find a violation, solve Question B, request flow entries for Question D and print out these flow entries before the flow for the violation packet was created and sent to the device by **l2_learning**. Without a reliable way to test the filter, it was re-

moved and we printed out each flow entry on the device. To solve this, the tool needs a way to time this correctly and request flow entries from a violating device after the violating flow has been created. Until then we risk to request flow entries for empty flow tables at these devices.

Another issue discovered on this level was that packets created by the **automator.py** would not create flow entries. We need the mac address to create flow entries on the switches, whereas the policy describing language only assigned IP addresses to host. To rectify this, we used the general mac broadcasting address "ff:ff:ff:ff:ff:ff" when creating the Ethernet payload at the **automator.py** before sending the packet. This issue could also be solved by assigning the mac together with the IP address on the policy description, but it was discovered too late with too little time allocated for fixing the bug.

7.1.4 Overall Troubleshooting

With the timing right at Question D we could tell the user violating policy rule, packets, packet path, devices, ports and flows, but the timing issue at requesting flow made displaying the violating flows unreliable.

The prototype can be used by network operators to monitor network wide traffic; to see if certain constraints and conditions are true, such as connections restricted by time or date, or that hosts don't transmit or receive more data then allowed by the policy. The source code is provided at a public Github repo: **https://github.com/HarisSistek/public_sdntroubleshoot** and also found in the appendix.

With time, difficulties with the approach was discovered, which resulted in a change from the original approach. As it often is with new technologies and tools, the access to documentation and proper use, in the form as wikies and online API resources, are lacking. The technical difficulties of learning a new API and using other SDN troubleshooting tools without much documentation to help, forced me to scale down the ambition and scope of the thesis.

7.2 Limitations

There was too little time allocated to look into equivalence checks for Question C and E. Checking if physical view matches device state or logical view matches physical view, would be great additions to the thesis, and would expand the usability of the prototype. We also covered a very small sample size of what constitutes a network policy. The prototype only covers conditions and constraints related to time, date and data. Expanding the descriptive policy language to cover more policy domains would have strengthened the thesis in this regard.

The technical difficulty in both learning and modifying tools such as ATPG and Hassel, without any documentation was a too big hurdle to overcome. With time, this changed the approach, scope and ambition of the thesis.

The prototype wasn't tested on mixed network. Knowing how the prototype would react when run on a virtual network interfacing with a physical.

7.3 Further Work

In order to complement, or build upon the work and results achieved in this thesis, there are a number of aspects which could be covered for further work. I have listed some below:

Expanding the descriptive policy language to handle a bigger sample of the policy domain. Policy rules that cover VLANs, topology, routing, etc. Expanding the policy description of hosts to also contain mac address, port number and network device to better automate packet creation and sending. Research and implement a better way to time the extraction of flow entries for Question D. As stated before, we need to find a way to extract flow stats after the other controller applications have finished processing their event. Incorporate the use of Hassel to help solve Question B search for black holes, loops and disconnectivity issues. Expanding the prototype to contain equivalence checks for Question C and E. With these questions added and researched, we would have completed Heller's troubleshooting decision tree. Finally, testing the prototype on mixed networks. Experiment how **sdn_dump** would react when we have one part virtual and physical network interfacing with each other.

Chapter 8

Conclusion

The main goal of this thesis work was to use the programmability of SDN and Heller's systematic approach to troubleshoot software defined network, in practice to automate the troubleshooting of a network. Based on the above mentioned goal, the problem statement was defined as: *How can we automatically troubleshoot networks using the layered structure of SDN?* We address it by developing a prototype that automatically moves down Heller's decision tree, while it is creating it's own testing behaviour.

The results showed that by the end of a troubleshooting search, the user would know what violating packet, packet path, policy rule, devices, flow entries and ports were the cause of the network problem. The results also showed that the automation of network behaviour could cause unexpected behaviour and that the issue of timing between different controller applications created instances with incomplete troubleshooting results.

The prototype showed that we could see network wide traffic and find policy violations, but that certain aspects remain to be addressed. Expanding the prototype to include Questions C and E, and expanding the scope of the descriptive language should be a priority as future work of this thesis, as this would complete Heller's decision tree.

Bibliography

[1] Hongyi Zeng et al. "Automatic test packet generation." In: *Proceedings of the 8th international conference on Emerging networking experiments and technologies*. ACM. 2012, pp. 241–252.

[2] Brandon Heller et al. "Leveraging SDN layering to systematically troubleshoot networks." In: *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*. ACM. 2013, pp. 37–42.

[3] Nikhil Handigol et al. "Where is the debugger for my software-defined network?" In: *Proceedings of the first workshop on Hot topics in software defined networks*. ACM. 2012, pp. 55–60.

[4] howstuffworks. *What is a packet?* URL: http://computer.howstuffworks.com/question5251.htm (visited on 04/12/2015).

[5] Ron Pacchiano. *The Difference Between Hubs, Switches and Routers*. URL: http://www.webopedia.com/DidYouKnow/Hardware_Software/router_switch_hub.asp (visited on 04/12/2015).

[6] Microsoft. *Microsoft Dev Net: Network Policies*. URL: https://msdn.microsoft.com/en-us/library/cc754107.aspx (visited on 04/20/2015).

[7] Nick Feamster, Jennifer Rexford, and Ellen Zegura. "The road to sdn: an intellectual history of programmable networks." In: *ACM SIGCOMM Computer Communication Review* 44.2 (2014), pp. 87–98.

[8] Thomas A Limoncelli. "Openflow: a radical new idea in networking." In: *Queue* 10.6 (2012), p. 40.

[9] Martin Casado et al. "Fabric: a retrospective on evolving SDN." In: *Proceedings of the first workshop on Hot topics in software defined networks*. ACM. 2012, pp. 85–90.

[10] Teemu Koponen Nick McKeown Martin Casado and Scott Shenker. *Software Defined Networks (SDN) slides*. URL: http://www.itc23.com/fileadmin/ITC23_files/slides/K1_McKeown-ITC_Keynote_Sept_2011.pdf (visited on 03/18/2015).

[11] Nick McKeown. *How SDN will Shape Networking - Nick McKeown*. URL: https://www.youtube.com/watch?v=c9-K5O_qYgA (visited on 03/18/2015).

[12] Wiki collaboration. *POX Wiki*. URL: https://openflow.stanford.edu/display/ONL/POX+Wiki (visited on 03/15/2015).

[13] The Open Networking Foundation. *OpenFlow Switch Specification v1.3.3*. URL: https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.3.3.pdf (visited on 03/18/2015).

[14] Brent Salisbury. *OpenFlow: Proactive vs Reactive Flows*. URL: http://networkstatic.net/openflow-proactive-vs-reactive-flows/ (visited on 02/21/2015).

[15] Mininet Team. *Mininet Overview*. URL: http://mininet.org/overview/ (visited on 03/17/2015).

[16] Mininet Team. *Download/Get Started With Mininet*. URL: http://mininet.org/download/ (visited on 03/17/2015).

[17] NOXRepo.org. *POX About*. URL: http://www.noxrepo.org/pox/about-pox/ (visited on 03/15/2015).

[18] NOXRepo.org. *NOX About*. URL: http://www.noxrepo.org/nox/about-nox/ (visited on 03/15/2015).

[19] NOXRepo.org. *NOX Classic*. URL: https://github.com/noxrepo/nox-classic/wiki (visited on 03/15/2015).

[20] die.net Linux Man Pages. *gdb(1) - Linux man page*. URL: http://linux.die.net/man/1/gdb (visited on 04/21/2015).

[21] die.net Linux Man Pages. *TCPDUMP*. URL: http://linux.die.net/man/8/tcpdump (visited on 04/12/2015).

[22] die.net Linux Man Pages. *ping(8) - Linux man page*. URL: http://linux.die.net/man/8/ping (visited on 04/12/2015).

[23] die.net Linux Man Pages. *traceroute(8) - Linux man page*. URL: http://linux.die.net/man/8/traceroute (visited on 04/12/2015).

[24] Andreas Wundsam et al. "OFRewind: Enabling Record and Replay Troubleshooting for Networks." In: *USENIX Annual Technical Conference*. 2011.

[25] Haohui Mai et al. "Debugging the data plane with anteater." In: *ACM SIGCOMM Computer Communication Review* 41.4 (2011), pp. 290–301.

[26] Peyman Kazemian, George Varghese, and Nick McKeown. "Header Space Analysis: Static Checking for Networks." In: *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*. San Jose, CA: USENIX, 2012, pp. 113–126. ISBN: 978-931971-92-8. URL: https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/kazemian.

[27] Nick Feamster. *Data-Plane Verification: Header Space Analysis*. URL: https://www.youtube.com/watch?v=xwhUh5PFGEs (visited on 04/23/2015).

[28] Wikipedia. *Switch Loop*. URL: http://en.wikipedia.org/wiki/Switching_loop (visited on 05/25/2015).

[29] OmniSecu. *What is Layer 2 Switching loop*. URL: http://www.omnisecu.com/cisco-certified-network-associate-ccna/what-is-layer-2-switching-loop.php (visited on 05/25/2015).

[30] OmniSecu. *What is Spanning Tree Protocol (STP)*. URL: http://www.omnisecu.com/cisco-certified-network-associate-ccna/what-is-spanning-tree-protocol-stp.php (visited on 05/25/2015).

[31] Lucas Brasilino Sandesh Shrestha and Murphy McCauley. *Re: [pox-dev] Create ICMP packets in POX*. URL: https://www.mail-archive.com/pox-dev@lists.noxrepo.org/msg01715.html (visited on 04/20/2015).

[32] Silvia Fichera and Murphy McCauley. *Re: [pox-dev] Create TCP packet*. URL: https://www.mail-archive.com/pox-dev@lists.noxrepo.org/msg00977.html (visited on 04/20/2015).

[33] Murphy McCauley (MurphyMc). *noxrepo/pox*. URL: https://github.com/noxrepo/pox/tree/carp/pox/forwarding (visited on 04/20/2015).

Appendices

Appendix A

Public GitHub Repo

Public GitHub repository: **https://github.com/HarisSistek/public_sdntroubleshoot**

Appendix B

Experiment Transcripts

B.1 ping-time-violation-found.txt

```
1  mininet@mininet-vm:~/pox$ ./pox.py pox.forwarding.
       l2_learning pox.sdntroubleshoot.sdn_dump --switch=2
       --mode=2
2  POX 0.2.0 (carp) / Copyright 2011-2013 James McCauley,
       et al.
3  Number of switches expected: 2
4  Searching for forwarding loops with spanning tree...
5  ### Stage 1: Started ###
6  Searching Question A: 'Does policy match actual
       behaviour?'
7  INFO:core:POX 0.2.0 (carp) is up.
8  INFO:openflow.of_01:[00-00-00-00-00-02 1] connected
9  INFO:openflow.of_01:[00-00-00-00-00-01 2] connected
10 Mon 15:40:39 DevID: 00-00-00-00-00-01 ARP: pkt e2:79:
       a7:67:cf:82 > ff:ff:ff:ff:ff:ff, hw e2:79:a7:67:cf
       :82 > 00:00:00:00:00:00: 1:REQUEST
11 Mon 15:40:39 DevID: 00-00-00-00-00-02 ARP: pkt e2:79:
       a7:67:cf:82 > ff:ff:ff:ff:ff:ff, hw e2:79:a7:67:cf
       :82 > 00:00:00:00:00:00: 1:REQUEST
12 Mon 15:40:39 DevID: 00-00-00-00-00-02 ARP: pkt 86:04:
       a1:72:22:af > e2:79:a7:67:cf:82, hw 86:04:a1:72:22:
       af > e2:79:a7:67:cf:82: 2:REPLY
13 Mon 15:40:39 DevID: 00-00-00-00-00-01 ARP: pkt 86:04:
       a1:72:22:af > e2:79:a7:67:cf:82, hw 86:04:a1:72:22:
       af > e2:79:a7:67:cf:82: 2:REPLY
14 Mon 15:40:39 DevID: 00-00-00-00-00-01 ICMP: pkt e2:79:
       a7:67:cf:82 > 86:04:a1:72:22:af, ip 10.0.0.1 >
       10.0.0.2: 8:ECHO_REQUEST
15 Mon 15:40:39 DevID: 00-00-00-00-00-02 ICMP: pkt e2:79:
       a7:67:cf:82 > 86:04:a1:72:22:af, ip 10.0.0.1 >
       10.0.0.2: 8:ECHO_REQUEST
16 ## Policy Violation Found: PACKET FORWARDING ##
```

```
17 > Packet: Mon 15:40:39 DevID: 00-00-00-00-00-01 ICMP:
       pkt e2:79:a7:67:cf:82 > 86:04:a1:72:22:af, ip
       10.0.0.1 > 10.0.0.2: 8:ECHO_REQUEST
18 > Packet: Mon 15:40:39 DevID: 00-00-00-00-00-02 ICMP:
       pkt e2:79:a7:67:cf:82 > 86:04:a1:72:22:af, ip
       10.0.0.1 > 10.0.0.2: 8:ECHO_REQUEST
19   violates:
20 > Rule: Time 10:00 to 23:00 block host1 to host2
21
22 ### Stage 2: Started ###
23 Question A: No, Policy does not match actual behaviour
24 Searching Question B: 'Does␣policy␣match␣device␣state?
       '
25 Checking switch connectivity:
26 Switch [00-00-00-00-00-01 2] is alive
27 Switch [00-00-00-00-00-02 1] is alive
28 0 device(s) unaccounted for
29 ### Stage 3: Started ###
30 Question B: Yes, Device state matches Policy.
31 Searching  Question D: 'Does␣device␣state␣match␣
       hardware?'
32 Requesting flow table entries from device
       [00-00-00-00-00-01 2]
33 Requesting flow table entries from device
       [00-00-00-00-00-02 1]
34 ## Flow Table for Deivce: 2 ##
35 > Entry 1 Match:
36 {'dl_type': 'ARP', 'nw_dst': '10.0.0.1/32', 'dl_src':
       '86:04:a1:72:22:af', 'nw_proto': 2, 'dl_dst': 'e2
       :79:a7:67:cf:82', 'dl_vlan': 65535, 'nw_src':
       IPAddr('10.0.0.2'), 'in_port': 1}
37 >> Entry Action
38 [{'max_len': 0, 'type': 'OFPAT_OUTPUT', 'port': 2}]
39 >> Byte Count
40 42
41 > Entry 2 Match:
42 {'dl_type': 'LLDP', 'dl_dst': '01:23:20:00:00:01'}
43 >> Entry Action
44 [{'max_len': 65535, 'type': 'OFPAT_OUTPUT', 'port': '
       OFPP_CONTROLLER'}]
45 >> Byte Count
46 0
47 ## Flow Table for Deivce: 1 ##
48 > Entry 1 Match:
49 {'dl_type': 'ARP', 'nw_dst': '10.0.0.1/32', 'dl_src':
       '86:04:a1:72:22:af', 'nw_proto': 2, 'dl_dst': 'e2
       :79:a7:67:cf:82', 'dl_vlan': 65535, 'nw_src':
       IPAddr('10.0.0.2'), 'in_port': 2}
```

```
50 >> Entry Action
51 [{'max_len': 0, 'type': 'OFPAT_OUTPUT', 'port': 1}]
52 >> Byte Count
53 42
54 > Entry 2 Match:
55 {'dl_type': 'LLDP', 'dl_dst': '01:23:20:00:00:01'}
56 >> Entry Action
57 [{'max_len': 65535, 'type': 'OFPAT_OUTPUT', 'port': '
      OFPP_CONTROLLER'}]
58 >> Byte Count
59 0
60 > Entry 3 Match:
61 {'dl_type': 'IP', 'nw_dst': '10.0.0.2/32', 'dl_src': '
      e2:79:a7:67:cf:82', 'nw_proto': 1, 'nw_tos': 0, '
      tp_dst': 0, 'tp_src': 8, 'dl_dst': '86:04:a1:72:22:
      af', 'dl_vlan': 65535, 'nw_src': IPAddr('10.0.0.1')
      , 'in_port': 1}
62 >> Entry Action
63 [{'max_len': 0, 'type': 'OFPAT_OUTPUT', 'port': 2}]
64 >> Byte Count
65 98
66 INFO:openflow.discovery:link detected:
      00-00-00-00-00-02.2 -> 00-00-00-00-00-01.2
67 INFO:openflow.discovery:link detected:
      00-00-00-00-00-01.2 -> 00-00-00-00-00-02.2
68 INFO:openflow.spanning_tree:4 ports changed
```

B.2 tcp-date-violation-found.txt

```
1  INFO:core:Down.
2  mininet@mininet-vm:~/pox$ ./pox.py pox.forwarding.
      l2_learning pox.sdntroubleshoot.sdn_dump --switch=2
       --mode=2
3  POX 0.2.0 (carp) / Copyright 2011-2013 James McCauley,
       et al.
4  Number of switches expected: 2
5  Searching for forwarding loops with spanning tree...
6  ### Stage 1: Started ###
7  Searching Question A: 'Does␣policy␣match␣actual␣
      behaviour?'
8  INFO:core:POX 0.2.0 (carp) is up.
9  INFO:openflow.of_01:[00-00-00-00-00-02 1] connected
10 INFO:openflow.of_01:[00-00-00-00-00-01 2] connected
11 INFO:openflow.discovery:link detected:
      00-00-00-00-00-02.2 -> 00-00-00-00-00-01.2
12 Sun 16:53:44 DevID: 00-00-00-00-00-01 TCP: pkt aa:46:
      cb:d3:9d:c9 > c6:80:16:13:f2:79, ip 10.0.0.1 >
      10.0.0.2: srcp 58421 dstp 5001, seq 697520508, ack
```

```
    0, flags 2
13 Sun 16:53:44 DevID: 00-00-00-00-00-02 TCP: pkt aa:46:
    cb:d3:9d:c9 > c6:80:16:13:f2:79, ip 10.0.0.1 >
    10.0.0.2: srcp 58421 dstp 5001, seq 697520508, ack
    0, flags 2
14 ## Policy Violation Found: PACKET FORWARDING ##
15 > Packet: Sun 16:53:44 DevID: 00-00-00-00-00-01 TCP:
    pkt aa:46:cb:d3:9d:c9 > c6:80:16:13:f2:79, ip
    10.0.0.1 > 10.0.0.2: srcp 58421 dstp 5001, seq
    697520508, ack 0, flags 2
16 > Packet: Sun 16:53:44 DevID: 00-00-00-00-00-02 TCP:
    pkt aa:46:cb:d3:9d:c9 > c6:80:16:13:f2:79, ip
    10.0.0.1 > 10.0.0.2: srcp 58421 dstp 5001, seq
    697520508, ack 0, flags 2
17   violates:
18 > Rule: Date MON to SUN block host1 to host2
19
20 ### Stage 2: Started ###
21 Question A: No, Policy does not match actual behaviour
22 Searching Question B: 'Does␣policy␣match␣device␣state?
    '
23 Checking switch connectivity:
24 Switch [00-00-00-00-00-01 2] is alive
25 Switch [00-00-00-00-00-02 1] is alive
26 0 device(s) unaccounted for
27 ### Stage 3: Started ###
28 Question B: Yes, Device state matches Policy.
29 Searching  Question D: 'Does␣device␣state␣match␣
    hardware?'
30 Requesting flow table entries from device
    [00-00-00-00-00-01 2]
31 Requesting flow table entries from device
    [00-00-00-00-00-02 1]
32 ## Flow Table for Deivce: 2 ##
33 > Entry 1 Match:
34 {'dl_type': 'LLDP', 'dl_dst': '01:23:20:00:00:01'}
35 >> Entry Action
36 [{'max_len': 65535, 'type': 'OFPAT_OUTPUT', 'port': '
    OFPP_CONTROLLER'}]
37 >> Byte Count
38 0
39 ## Flow Table for Deivce: 1 ##
40 > Entry 1 Match:
41 {'dl_type': 'LLDP', 'dl_dst': '01:23:20:00:00:01'}
42 >> Entry Action
43 [{'max_len': 65535, 'type': 'OFPAT_OUTPUT', 'port': '
    OFPP_CONTROLLER'}]
44 >> Byte Count
```

45 41

B.3 ping-data-violation-found.txt

```
1  mininet@mininet-vm:~/pox$ ./pox.py pox.forwarding.
       l2_learning pox.sdntroubleshoot.sdn_dump --switch=2
       --mode=2
2  POX 0.2.0 (carp) / Copyright 2011-2013 James McCauley,
       et al.
3  Number of switches expected: 2
4  Searching for forwarding loops with spanning tree...
5  ### Stage 1: Started ###
6  Searching Question A: 'Does␣policy␣match␣actual␣
       behaviour?'
7  INFO:core:POX 0.2.0 (carp) is up.
8  INFO:openflow.of_01:[00-00-00-00-00-02 1] connected
9  INFO:openflow.of_01:[00-00-00-00-00-01 2] connected
10 INFO:openflow.discovery:link detected:
       00-00-00-00-00-02.2 -> 00-00-00-00-00-01.2
11 Sun 16:56:17 DevID: 00-00-00-00-00-01 ICMP: pkt aa:46:
       cb:d3:9d:c9 > c6:80:16:13:f2:79, ip 10.0.0.1 >
       10.0.0.2: 8:ECHO_REQUEST
12 Sun 16:56:17 DevID: 00-00-00-00-00-02 ICMP: pkt aa:46:
       cb:d3:9d:c9 > c6:80:16:13:f2:79, ip 10.0.0.1 >
       10.0.0.2: 8:ECHO_REQUEST
13 Sun 16:56:17 DevID: 00-00-00-00-00-02 ICMP: pkt c6
       :80:16:13:f2:79 > aa:46:cb:d3:9d:c9, ip 10.0.0.2 >
       10.0.0.1: 0:ECHO_REPY
14 Sun 16:56:17 DevID: 00-00-00-00-00-01 ICMP: pkt c6
       :80:16:13:f2:79 > aa:46:cb:d3:9d:c9, ip 10.0.0.2 >
       10.0.0.1: 0:ECHO_REPY
15 Sun 16:56:18 DevID: 00-00-00-00-00-01 ICMP: pkt aa:46:
       cb:d3:9d:c9 > c6:80:16:13:f2:79, ip 10.0.0.1 >
       10.0.0.2: 8:ECHO_REQUEST
16 ## Policy Violation Found: SENDING TO MUCH DATA ##
17 > IP: 10.0.0.2 address sent: 21503948 bytes
18 > IP: 10.0.0.2 can only send: 1000 bytes
19   violates:
20 > Rule: Data 1000 b from host2
21
22 ### Stage 2: Started ###
23 Question A: No, Policy does not match actual behaviour
24 Searching Question B: 'Does␣policy␣match␣device␣state?
       '
25 Checking switch connectivity:
26 Switch [00-00-00-00-00-01 2] is alive
27 Switch [00-00-00-00-00-02 1] is alive
28 0 device(s) unaccounted for
```

```
29 ### Stage 3: Started ###
30 Question B: Yes, Device state matches Policy.
31 Searching Question D: 'Does␣device␣state␣match␣
      hardware?'
32 Requesting flow table entries from device
      [00-00-00-00-00-01 2]
33 Requesting flow table entries from device
      [00-00-00-00-00-02 1]
34 ## Flow Table for Deivce: 2 ##
35 > Entry 1 Match:
36 {'dl_type': 'LLDP', 'dl_dst': '01:23:20:00:00:01'}
37 >> Entry Action
38 [{'max_len': 65535, 'type': 'OFPAT_OUTPUT', 'port': '
      OFPP_CONTROLLER'}]
39 >> Byte Count
40 0
41 > Entry 2 Match:
42 {'dl_type': 'IP', 'nw_dst': '10.0.0.1/32', 'dl_src': '
      c6:80:16:13:f2:79', 'nw_proto': 1, 'nw_tos': 0, '
      tp_dst': 0, 'tp_src': 0, 'dl_dst': 'aa:46:cb:d3:9d:
      c9', 'dl_vlan': 65535, 'nw_src': IPAddr('10.0.0.2')
      , 'in_port': 1}
43 >> Entry Action
44 [{'max_len': 0, 'type': 'OFPAT_OUTPUT', 'port': 2}]
45 >> Byte Count
46 98
47 INFO:openflow.discovery:link detected:
      00-00-00-00-00-01.2 -> 00-00-00-00-00-02.2
48 INFO:openflow.spanning_tree:4 ports changed
49 ## Flow Table for Deivce: 1 ##
50 > Entry 1 Match:
51 {'dl_type': 'LLDP', 'dl_dst': '01:23:20:00:00:01'}
52 >> Entry Action
53 [{'max_len': 65535, 'type': 'OFPAT_OUTPUT', 'port': '
      OFPP_CONTROLLER'}]
54 >> Byte Count
55 41
56 > Entry 2 Match:
57 {'dl_type': 'IP', 'nw_dst': '10.0.0.1/32', 'dl_src': '
      c6:80:16:13:f2:79', 'nw_proto': 1, 'nw_tos': 0, '
      tp_dst': 0, 'tp_src': 0, 'dl_dst': 'aa:46:cb:d3:9d:
      c9', 'dl_vlan': 65535, 'nw_src': IPAddr('10.0.0.2')
      , 'in_port': 2}
58 >> Entry Action
59 [{'max_len': 0, 'type': 'OFPAT_OUTPUT', 'port': 1}]
60 >> Byte Count
61 196
62 > Entry 3 Match:
```

```
63 {'dl_type': 'IP', 'nw_dst': '10.0.0.2/32', 'dl_src': '
      aa:46:cb:d3:9d:c9', 'nw_proto': 1, 'nw_tos': 0, '
      tp_dst': 0, 'tp_src': 8, 'dl_dst': 'c6:80:16:13:f2
      :79', 'dl_vlan': 65535, 'nw_src': IPAddr('10.0.0.1'
      ), 'in_port': 1}
64 >> Entry Action
65 [{'max_len': 0, 'type': 'OFPAT_OUTPUT', 'port': 2}]
66 >> Byte Count
67 98
68 ^CINFO:core:Going down...
69 INFO:openflow.of_01:[00-00-00-00-00-01 2] disconnected
70 INFO:openflow.of_01:[00-00-00-00-00-02 1] disconnected
71 INFO:core:Down.
```

B.4 ping-date-violation-found.txt

```
 1 mininet@mininet-vm:~/pox$ ./pox.py pox.forwarding.
      l2_learning pox.sdntroubleshoot.sdn_dump --switch=2
       --mode=2
 2 POX 0.2.0 (carp) / Copyright 2011-2013 James McCauley,
       et al.
 3 Number of switches expected: 2
 4 Searching for forwarding loops with spanning tree...
 5 ### Stage 1: Started ###
 6 Searching Question A: 'Does␣policy␣match␣actual␣
      behaviour?'
 7 INFO:core:POX 0.2.0 (carp) is up.
 8 INFO:openflow.of_01:[00-00-00-00-00-02 1] connected
 9 INFO:openflow.of_01:[00-00-00-00-00-01 2] connected
10 INFO:openflow.discovery:link detected:
      00-00-00-00-00-02.2 -> 00-00-00-00-00-01.2
11 INFO:sdntroubleshoot.sdn_dump:UNKNOWN packet c6
      :80:16:13:f2:79
12 INFO:sdntroubleshoot.sdn_dump:UNKNOWN packet c6
      :80:16:13:f2:79
13 INFO:sdntroubleshoot.sdn_dump:UNKNOWN packet aa:46:cb:
      d3:9d:c9
14 INFO:sdntroubleshoot.sdn_dump:UNKNOWN packet aa:46:cb:
      d3:9d:c9
15 INFO:openflow.discovery:link detected:
      00-00-00-00-00-01.2 -> 00-00-00-00-00-02.2
16 INFO:openflow.spanning_tree:4 ports changed
17 Sun 16:52:43 DevID: 00-00-00-00-00-01 ARP: pkt aa:46:
      cb:d3:9d:c9 > ff:ff:ff:ff:ff:ff, hw aa:46:cb:d3:9d:
      c9 > 00:00:00:00:00:00: 1:REQUEST
18 Sun 16:52:43 DevID: 00-00-00-00-00-02 ARP: pkt aa:46:
      cb:d3:9d:c9 > ff:ff:ff:ff:ff:ff, hw aa:46:cb:d3:9d:
      c9 > 00:00:00:00:00:00: 1:REQUEST
```

```
19 Sun 16:52:43 DevID: 00-00-00-00-00-02 ARP: pkt c6
       :80:16:13:f2:79 > aa:46:cb:d3:9d:c9, hw c6
       :80:16:13:f2:79 > aa:46:cb:d3:9d:c9: 2:REPLY
20 Sun 16:52:43 DevID: 00-00-00-00-00-01 ARP: pkt c6
       :80:16:13:f2:79 > aa:46:cb:d3:9d:c9, hw c6
       :80:16:13:f2:79 > aa:46:cb:d3:9d:c9: 2:REPLY
21 Sun 16:52:43 DevID: 00-00-00-00-00-01 ICMP: pkt aa:46:
       cb:d3:9d:c9 > c6:80:16:13:f2:79, ip 10.0.0.1 >
       10.0.0.2: 8:ECHO_REQUEST
22 Sun 16:52:43 DevID: 00-00-00-00-00-02 ICMP: pkt aa:46:
       cb:d3:9d:c9 > c6:80:16:13:f2:79, ip 10.0.0.1 >
       10.0.0.2: 8:ECHO_REQUEST
23 ## Policy Violation Found: PACKET FORWARDING ##
24 > Packet: Sun 16:52:43 DevID: 00-00-00-00-00-01 ICMP:
       pkt aa:46:cb:d3:9d:c9 > c6:80:16:13:f2:79, ip
       10.0.0.1 > 10.0.0.2: 8:ECHO_REQUEST
25 > Packet: Sun 16:52:43 DevID: 00-00-00-00-00-02 ICMP:
       pkt aa:46:cb:d3:9d:c9 > c6:80:16:13:f2:79, ip
       10.0.0.1 > 10.0.0.2: 8:ECHO_REQUEST
26   violates:
27 > Rule: Date MON to SUN block host1 to host2
28
29 ### Stage 2: Started ###
30 Question A: No, Policy does not match actual behaviour
31 Searching Question B: 'Does␣policy␣match␣device␣state?
       '
32 Checking switch connectivity:
33 Switch [00-00-00-00-00-01 2] is alive
34 Switch [00-00-00-00-00-02 1] is alive
35 0 device(s) unaccounted for
36 ### Stage 3: Started ###
37 Question B: Yes, Device state matches Policy.
38 Searching Question D: 'Does␣device␣state␣match␣
       hardware?'
39 Requesting flow table entries from device
       [00-00-00-00-00-01 2]
40 Requesting flow table entries from device
       [00-00-00-00-00-02 1]
41 ## Flow Table for Deivce: 2 ##
42 > Entry 1 Match:
43 {'dl_type': 'ARP', 'nw_dst': '10.0.0.1/32', 'dl_src':
       'c6:80:16:13:f2:79', 'nw_proto': 2, 'dl_dst': 'aa
       :46:cb:d3:9d:c9', 'dl_vlan': 65535, 'nw_src':
       IPAddr('10.0.0.2'), 'in_port': 1}
44 >> Entry Action
45 [{'max_len': 0, 'type': 'OFPAT_OUTPUT', 'port': 2}]
46 >> Byte Count
47 42
```

```
48 > Entry 2 Match:
49 {'dl_type': 'LLDP', 'dl_dst': '01:23:20:00:00:01'}
50 >> Entry Action
51 [{'max_len': 65535, 'type': 'OFPAT_OUTPUT', 'port': '
      OFPP_CONTROLLER'}]
52 >> Byte Count
53 41
54 ## Flow Table for Deivce: 1 ##
55 > Entry 1 Match:
56 {'dl_type': 'ARP', 'nw_dst': '10.0.0.1/32', 'dl_src':
      'c6:80:16:13:f2:79', 'nw_proto': 2, 'dl_dst': 'aa
      :46:cb:d3:9d:c9', 'dl_vlan': 65535, 'nw_src':
      IPAddr('10.0.0.2'), 'in_port': 2}
57 >> Entry Action
58 [{'max_len': 0, 'type': 'OFPAT_OUTPUT', 'port': 1}]
59 >> Byte Count
60 42
61 > Entry 2 Match:
62 {'dl_type': 'LLDP', 'dl_dst': '01:23:20:00:00:01'}
63 >> Entry Action
64 [{'max_len': 65535, 'type': 'OFPAT_OUTPUT', 'port': '
      OFPP_CONTROLLER'}]
65 >> Byte Count
66 123
67 > Entry 3 Match:
68 {'dl_type': 'IP', 'nw_dst': '10.0.0.2/32', 'dl_src': '
      aa:46:cb:d3:9d:c9', 'nw_proto': 1, 'nw_tos': 0, '
      tp_dst': 0, 'tp_src': 8, 'dl_dst': 'c6:80:16:13:f2
      :79', 'dl_vlan': 65535, 'nw_src': IPAddr('10.0.0.1'
      ), 'in_port': 1}
69 >> Entry Action
70 [{'max_len': 0, 'type': 'OFPAT_OUTPUT', 'port': 2}]
71 >> Byte Count
72 98
```

B.5 auto-ping-reply.txt

```
1 mininet@mininet-vm:~/pox$ ./pox.py pox.forwarding.
     l2_learning pox.sdntroubleshoot.sdn_dump --switch=1
      --mode=3
2 POX 0.2.0 (carp) / Copyright 2011-2013 James McCauley,
      et al.
3 Number of switches expected: 1
4 Searching for forwarding loops with spanning tree...
5 match1
6 ### Stage 1: Started ###
7 Searching Question A: 'Does␣policy␣match␣actual␣
     behaviour?'
```

```
 8 INFO:core:POX 0.2.0 (carp) is up.
 9 INFO:openflow.of_01:[00-00-00-00-00-01 1] connected
10 Sun 17:07:12 DevID: 00-00-00-00-00-01 ICMP: pkt ff:ff:
      ff:ff:ff:ff > ff:ff:ff:ff:ff:ff, ip 10.0.0.1 >
      10.0.0.2: 8:ECHO_REQUEST
11 Sun 17:07:12 DevID: 00-00-00-00-00-01 ARP: pkt fe:67:
      e8:4c:2e:12 > ff:ff:ff:ff:ff:ff, hw fe:67:e8:4c:2e
      :12 > 00:00:00:00:00:00: 1:REQUEST
12 Sun 17:07:12 DevID: 00-00-00-00-00-01 ARP: pkt 4e:ce
      :20:54:1b:9a > fe:67:e8:4c:2e:12, hw 4e:ce:20:54:1b
      :9a > fe:67:e8:4c:2e:12: 2:REPLY
13 Sun 17:07:12 DevID: 00-00-00-00-00-01 ICMP: pkt fe:67:
      e8:4c:2e:12 > 4e:ce:20:54:1b:9a, ip 10.0.0.2 >
      10.0.0.1: 0:ECHO_REPY
14 ## Policy Violation Found: PACKET RETURN ##
15 > Packet: Sun 17:07:12 DevID: 00-00-00-00-00-01 ICMP:
      pkt ff:ff:ff:ff:ff:ff > ff:ff:ff:ff:ff:ff, ip
      10.0.0.1 > 10.0.0.2: 8:ECHO_REQUEST
16 > Packet: Sun 17:07:12 DevID: 00-00-00-00-00-01 ICMP:
      pkt fe:67:e8:4c:2e:12 > 4e:ce:20:54:1b:9a, ip
      10.0.0.2 > 10.0.0.1: 0:ECHO_REPY
17   violates:
18 > Rule: Time 10:00 to 23:00 block host1 to host2
19
20 ### Stage 2: Started ###
21 Question A: No, Policy does not match actual behaviour
22 Searching Question B: 'Does␣policy␣match␣device␣state?
      '
23 Checking switch connectivity:
24 Switch [00-00-00-00-00-01 1] is alive
25 0 device(s) unaccounted for
26 ### Stage 3: Started ###
27 Question B: Yes, Device state matches Policy.
28 Searching  Question D: 'Does␣device␣state␣match␣
      hardware?'
29 Requesting flow table entries from device
      [00-00-00-00-00-01 1]
30 ## Flow Table for Deivce: 1 ##
31 > Entry 1 Match:
32 {'dl_type': 'ARP', 'nw_dst': '10.0.0.2/32', 'dl_src':
      '4e:ce:20:54:1b:9a', 'nw_proto': 2, 'dl_dst': 'fe
      :67:e8:4c:2e:12', 'dl_vlan': 65535, 'nw_src':
      IPAddr('10.0.0.1'), 'in_port': 1}
33 >> Entry Action
34 [{'max_len': 0, 'type': 'OFPAT_OUTPUT', 'port': 2}]
35 >> Byte Count
36 42
37 > Entry 2 Match:
```

```
38 {'dl_type': 'LLDP', 'dl_dst': '01:23:20:00:00:01'}
39 >> Entry Action
40 [{'max_len': 65535, 'type': 'OFPAT_OUTPUT', 'port': '
     OFPP_CONTROLLER'}]
41 >> Byte Count
42 0
```

B.6 auto-tcp-forward.txt

```
 1 mininet@mininet-vm:~/pox$ ./pox.py pox.forwarding.
     l2_learning pox.sdntroubleshoot.sdn_dump --switch=2
      --mode=3
 2 POX 0.2.0 (carp) / Copyright 2011-2013 James McCauley,
      et al.
 3 Number of switches expected: 2
 4 Searching for forwarding loops with spanning tree...
 5 match1
 6 ### Stage 1: Started ###
 7 Searching Question A: 'Does␣policy␣match␣actual␣
     behaviour?'
 8 INFO:core:POX 0.2.0 (carp) is up.
 9 INFO:openflow.of_01:[00-00-00-00-00-02 1] connected
10 INFO:openflow.of_01:[00-00-00-00-00-01 2] connected
11 Sun 17:08:40 DevID: 00-00-00-00-00-01 TCP: pkt ff:ff:
     ff:ff:ff:ff > ff:ff:ff:ff:ff:ff, ip 10.0.0.1 >
     10.0.0.2: srcp 48966 dstp 5001, seq 0, ack 0, flags
      2
12 Sun 17:08:40 DevID: 00-00-00-00-00-02 TCP: pkt ff:ff:
     ff:ff:ff:ff > ff:ff:ff:ff:ff:ff, ip 10.0.0.1 >
     10.0.0.2: srcp 48966 dstp 5001, seq 0, ack 0, flags
      2
13 ## Policy Violation Found: PACKET FORWARDING ##
14 > Packet: Sun 17:08:40 DevID: 00-00-00-00-00-01 TCP:
     pkt ff:ff:ff:ff:ff:ff > ff:ff:ff:ff:ff:ff, ip
     10.0.0.1 > 10.0.0.2: srcp 48966 dstp 5001, seq 0,
     ack 0, flags 2
15 > Packet: Sun 17:08:40 DevID: 00-00-00-00-00-02 TCP:
     pkt ff:ff:ff:ff:ff:ff > ff:ff:ff:ff:ff:ff, ip
     10.0.0.1 > 10.0.0.2: srcp 48966 dstp 5001, seq 0,
     ack 0, flags 2
16   violates:
17 > Rule: Time 10:00 to 23:00 block host1 to host2 prot
     TCP
18
19 ### Stage 2: Started ###
20 Question A: No, Policy does not match actual behaviour
21 Searching Question B: 'Does␣policy␣match␣device␣state?
     '
```

```
22 Checking switch connectivity:
23 Switch [00-00-00-00-00-01 2] is alive
24 Switch [00-00-00-00-00-02 1] is alive
25 0 device(s) unaccounted for
26 ### Stage 3: Started ###
27 Question B: Yes, Device state matches Policy.
28 Searching Question D: 'Does device state match
       hardware?'
29 Requesting flow table entries from device
       [00-00-00-00-00-01 2]
30 Requesting flow table entries from device
       [00-00-00-00-00-02 1]
31 ## Flow Table for Deivce: 2 ##
32 > Entry 1 Match:
33 {'dl_type': 'LLDP', 'dl_dst': '01:23:20:00:00:01'}
34 >> Entry Action
35 [{'max_len': 65535, 'type': 'OFPAT_OUTPUT', 'port': '
       OFPP_CONTROLLER'}]
36 >> Byte Count
37 0
38 ## Flow Table for Deivce: 1 ##
39 > Entry 1 Match:
40 {'dl_type': 'LLDP', 'dl_dst': '01:23:20:00:00:01'}
41 >> Entry Action
42 [{'max_len': 65535, 'type': 'OFPAT_OUTPUT', 'port': '
       OFPP_CONTROLLER'}]
43 >> Byte Count
44 0
45 INFO:openflow.discovery:link detected:
       00-00-00-00-00-02.2 -> 00-00-00-00-00-01.2
46 INFO:openflow.discovery:link detected:
       00-00-00-00-00-01.2 -> 00-00-00-00-00-02.2
47 INFO:openflow.spanning_tree:4 ports changed
48 ^CINFO:core:Going down...
49 INFO:openflow.of_01:[00-00-00-00-00-01 2] disconnected
50 INFO:openflow.of_01:[00-00-00-00-00-02 1] disconnected
51 INFO:core:Down.
```

B.7 without-firewall.txt

```
1 mininet@mininet-vm:~/pox$ ./pox.py pox.forwarding.
       l2_learning pox.sdntroubleshoot.sdn_dump --switch=2
       --mode=3
2 POX 0.2.0 (carp) / Copyright 2011-2013 James McCauley,
       et al.
3 Number of switches expected: 2
4 Searching for forwarding loops with spanning tree...
5 match1
```

```
### Stage 1: Started ###
Searching Question A: 'Does policy match actual
    behaviour?'
INFO:core:POX 0.2.0 (carp) is up.
INFO:openflow.of_01:[00-00-00-00-00-02 1] connected
INFO:openflow.of_01:[00-00-00-00-00-01 2] connected
Sun 17:43:13 DevID: 00-00-00-00-00-01 TCP: pkt ff:ff:
    ff:ff:ff:ff > ff:ff:ff:ff:ff:ff, ip 10.0.0.1 >
    10.0.0.2: srcp 41238 dstp 5001, seq 0, ack 0, flags
     2
Sun 17:43:13 DevID: 00-00-00-00-00-02 TCP: pkt ff:ff:
    ff:ff:ff:ff > ff:ff:ff:ff:ff:ff, ip 10.0.0.1 >
    10.0.0.2: srcp 41238 dstp 5001, seq 0, ack 0, flags
     2
## Policy Violation Found: PACKET FORWARDING ##
> Packet: Sun 17:43:13 DevID: 00-00-00-00-00-01 TCP:
    pkt ff:ff:ff:ff:ff:ff > ff:ff:ff:ff:ff:ff, ip
    10.0.0.1 > 10.0.0.2: srcp 41238 dstp 5001, seq 0,
    ack 0, flags 2
> Packet: Sun 17:43:13 DevID: 00-00-00-00-00-02 TCP:
    pkt ff:ff:ff:ff:ff:ff > ff:ff:ff:ff:ff:ff, ip
    10.0.0.1 > 10.0.0.2: srcp 41238 dstp 5001, seq 0,
    ack 0, flags 2
   violates:
> Rule: Time 10:00 to 23:00 block host1 to host2 prot
    TCP sport 41238 dport 5001

### Stage 2: Started ###
Question A: No, Policy does not match actual behaviour
Searching Question B: 'Does policy match device state?
    '
Checking switch connectivity:
Switch [00-00-00-00-00-01 2] is alive
Switch [00-00-00-00-00-02 1] is alive
0 device(s) unaccounted for
### Stage 3: Started ###
Question B: Yes, Device state matches Policy.
Searching  Question D: 'Does device state match
    hardware?'
Requesting flow table entries from device
    [00-00-00-00-00-01 2]
Requesting flow table entries from device
    [00-00-00-00-00-02 1]
## Flow Table for Deivce: 2 ##
> Entry 1 Match:
{'dl_type': 'LLDP', 'dl_dst': '01:23:20:00:00:01'}
>> Entry Action
```

```
35 [{'max_len': 65535, 'type': 'OFPAT_OUTPUT', 'port': '
     OFPP_CONTROLLER'}]
36 >> Byte Count
37 0
38 ## Flow Table for Deivce: 1 ##
39 > Entry 1 Match:
40 {'dl_type': 'LLDP', 'dl_dst': '01:23:20:00:00:01'}
41 >> Entry Action
42 [{'max_len': 65535, 'type': 'OFPAT_OUTPUT', 'port': '
     OFPP_CONTROLLER'}]
43 >> Byte Count
44 0
45 INFO:openflow.discovery:link detected:
     00-00-00-00-00-02.2 -> 00-00-00-00-00-01.2
46 INFO:openflow.discovery:link detected:
     00-00-00-00-00-01.2 -> 00-00-00-00-00-02.2
47 INFO:openflow.spanning_tree:4 ports changed
48 ^CINFO:core:Going down...
49 INFO:openflow.of_01:[00-00-00-00-00-01 2] disconnected
50 INFO:openflow.of_01:[00-00-00-00-00-02 1] disconnected
51 INFO:core:Down.
```

B.8 with-firewall.txt

```
 1 mininet@mininet-vm:~/pox$ ./pox.py pox.forwarding.
     l2_learning pox.sdntroubleshoot.sdn_dump --switch=2
     --mode=3 pox.forwarding.blocker --ports=5001
 2 POX 0.2.0 (carp) / Copyright 2011-2013 James McCauley,
     et al.
 3 Number of switches expected: 2
 4 Searching for forwarding loops with spanning tree...
 5 match1
 6 ### Stage 1: Started ###
 7 Searching Question A: 'Does␣policy␣match␣actual␣
     behaviour?'
 8 INFO:core:POX 0.2.0 (carp) is up.
 9 INFO:openflow.of_01:[00-00-00-00-00-02 1] connected
10 INFO:openflow.of_01:[00-00-00-00-00-01 2] connected
11 Sun 17:44:16 DevID: 00-00-00-00-00-01 TCP: pkt ff:ff:
     ff:ff:ff:ff > ff:ff:ff:ff:ff:ff, ip 10.0.0.1 >
     10.0.0.2: srcp 41238 dstp 5001, seq 0, ack 0, flags
     2
12 INFO:openflow.discovery:link detected:
     00-00-00-00-00-02.2 -> 00-00-00-00-00-01.2
13 INFO:openflow.discovery:link detected:
     00-00-00-00-00-01.2 -> 00-00-00-00-00-02.2
14 INFO:openflow.spanning_tree:4 ports changed
15 ^CINFO:core:Going down...
```

```
16 INFO:openflow.of_01:[00-00-00-00-00-01 2] disconnected
17 INFO:openflow.of_01:[00-00-00-00-00-02 1] disconnected
18 INFO:core:Down.
```

B.9 mode1-10switch-ping.txt

```
 1 mininet@mininet-vm:~/pox$ ./pox.py pox.forwarding.
     l2_learning pox.sdntroubleshoot.sdn_dump --switch
     =10 --mode=1
 2 POX 0.2.0 (carp) / Copyright 2011-2013 James McCauley,
     et al.
 3 Number of switches expected: 10
 4 Searching for forwarding loops with spanning tree...
 5 ### Stage 1: Started ###
 6 Searching Question A: 'Does policy match actual
     behaviour?'
 7 INFO:core:POX 0.2.0 (carp) is up.
 8 INFO:openflow.of_01:[00-00-00-00-00-08 3] connected
 9 INFO:openflow.of_01:[00-00-00-00-00-09 4] connected
10 INFO:openflow.of_01:[00-00-00-00-00-02 2] connected
11 INFO:openflow.of_01:[00-00-00-00-00-01 5] connected
12 INFO:openflow.of_01:[00-00-00-00-00-05 1] connected
13 INFO:openflow.of_01:[00-00-00-00-00-03 6] connected
14 INFO:openflow.of_01:[00-00-00-00-00-04 7] connected
15 INFO:openflow.of_01:[00-00-00-00-00-07 8] connected
16 INFO:openflow.of_01:[00-00-00-00-00-06 9] connected
17 INFO:openflow.of_01:[00-00-00-00-00-0a 10] connected
18 INFO:openflow.discovery:link detected:
     00-00-00-00-00-08.3 -> 00-00-00-00-00-09.2
19 INFO:openflow.discovery:link detected:
     00-00-00-00-00-08.2 -> 00-00-00-00-00-07.3
20 INFO:openflow.discovery:link detected:
     00-00-00-00-00-09.3 -> 00-00-00-00-00-0a.2
21 INFO:sdntroubleshoot.sdn_dump:UNKNOWN packet 6a:51:79:
     b4:22:e0
22 INFO:sdntroubleshoot.sdn_dump:UNKNOWN packet 6a:51:79:
     b4:22:e0
23 INFO:sdntroubleshoot.sdn_dump:UNKNOWN packet 6a:51:79:
     b4:22:e0
24 INFO:sdntroubleshoot.sdn_dump:UNKNOWN packet 6a:51:79:
     b4:22:e0
25 INFO:sdntroubleshoot.sdn_dump:UNKNOWN packet 6a:51:79:
     b4:22:e0
26 INFO:openflow.discovery:link detected:
     00-00-00-00-00-09.2 -> 00-00-00-00-00-08.3
27 INFO:openflow.spanning_tree:6 ports changed
28 INFO:sdntroubleshoot.sdn_dump:UNKNOWN packet 6a:51:79:
     b4:22:e0
```

```
29 INFO:sdntroubleshoot.sdn_dump:UNKNOWN packet 6a:51:79:
       b4:22:e0
30 INFO:sdntroubleshoot.sdn_dump:UNKNOWN packet 6a:51:79:
       b4:22:e0
31 INFO:sdntroubleshoot.sdn_dump:UNKNOWN packet 6a:51:79:
       b4:22:e0
32 INFO:sdntroubleshoot.sdn_dump:UNKNOWN packet b6
       :97:29:5e:02:8f
33 INFO:sdntroubleshoot.sdn_dump:UNKNOWN packet b6
       :97:29:5e:02:8f
34 INFO:sdntroubleshoot.sdn_dump:UNKNOWN packet b6
       :97:29:5e:02:8f
35 INFO:sdntroubleshoot.sdn_dump:UNKNOWN packet b6
       :97:29:5e:02:8f
36 INFO:sdntroubleshoot.sdn_dump:UNKNOWN packet b6
       :97:29:5e:02:8f
37 INFO:sdntroubleshoot.sdn_dump:UNKNOWN packet b6
       :97:29:5e:02:8f
38 INFO:sdntroubleshoot.sdn_dump:UNKNOWN packet b6
       :97:29:5e:02:8f
39 INFO:sdntroubleshoot.sdn_dump:UNKNOWN packet b6
       :97:29:5e:02:8f
40 INFO:sdntroubleshoot.sdn_dump:UNKNOWN packet b6
       :97:29:5e:02:8f
41 INFO:sdntroubleshoot.sdn_dump:UNKNOWN packet 76:8b:e8:
       c4:7e:ec
42 INFO:sdntroubleshoot.sdn_dump:UNKNOWN packet 76:8b:e8:
       c4:7e:ec
43 INFO:sdntroubleshoot.sdn_dump:UNKNOWN packet 76:8b:e8:
       c4:7e:ec
44 INFO:sdntroubleshoot.sdn_dump:UNKNOWN packet 76:8b:e8:
       c4:7e:ec
45 INFO:openflow.discovery:link detected:
       00-00-00-00-00-02.3 -> 00-00-00-00-00-03.2
46 INFO:sdntroubleshoot.sdn_dump:UNKNOWN packet 76:8b:e8:
       c4:7e:ec
47 INFO:sdntroubleshoot.sdn_dump:UNKNOWN packet 76:8b:e8:
       c4:7e:ec
48 INFO:sdntroubleshoot.sdn_dump:UNKNOWN packet 76:8b:e8:
       c4:7e:ec
49 INFO:sdntroubleshoot.sdn_dump:UNKNOWN packet 76:8b:e8:
       c4:7e:ec
50 INFO:sdntroubleshoot.sdn_dump:UNKNOWN packet 76:8b:e8:
       c4:7e:ec
51 INFO:sdntroubleshoot.sdn_dump:UNKNOWN packet 6a:4a:fb:
       c9:9b:4e
52 INFO:openflow.discovery:link detected:
       00-00-00-00-00-02.2 -> 00-00-00-00-00-01.2
```

```
53 INFO:sdntroubleshoot.sdn_dump:UNKNOWN packet 6a:4a:fb:
    c9:9b:4e
54 INFO:sdntroubleshoot.sdn_dump:UNKNOWN packet 4a:7f
    :78:2d:7d:fa
55 INFO:sdntroubleshoot.sdn_dump:UNKNOWN packet 4a:7f
    :78:2d:7d:fa
56 INFO:sdntroubleshoot.sdn_dump:UNKNOWN packet 4a:7f
    :78:2d:7d:fa
57 INFO:sdntroubleshoot.sdn_dump:UNKNOWN packet 4a:7f
    :78:2d:7d:fa
58 INFO:sdntroubleshoot.sdn_dump:UNKNOWN packet 4a:7f
    :78:2d:7d:fa
59 INFO:sdntroubleshoot.sdn_dump:UNKNOWN packet 4a:7f
    :78:2d:7d:fa
60 INFO:sdntroubleshoot.sdn_dump:UNKNOWN packet 4a:7f
    :78:2d:7d:fa
61 INFO:sdntroubleshoot.sdn_dump:UNKNOWN packet 4a:7f
    :78:2d:7d:fa
62 INFO:sdntroubleshoot.sdn_dump:UNKNOWN packet 4a:7f
    :78:2d:7d:fa
63 INFO:sdntroubleshoot.sdn_dump:UNKNOWN packet 66:9c:7f
    :16:8e:38
64 INFO:sdntroubleshoot.sdn_dump:UNKNOWN packet 66:9c:7f
    :16:8e:38
65 INFO:sdntroubleshoot.sdn_dump:UNKNOWN packet 66:9c:7f
    :16:8e:38
66 INFO:sdntroubleshoot.sdn_dump:UNKNOWN packet 66:9c:7f
    :16:8e:38
67 INFO:sdntroubleshoot.sdn_dump:UNKNOWN packet 66:9c:7f
    :16:8e:38
68 INFO:sdntroubleshoot.sdn_dump:UNKNOWN packet 66:9c:7f
    :16:8e:38
69 INFO:sdntroubleshoot.sdn_dump:UNKNOWN packet 66:9c:7f
    :16:8e:38
70 INFO:sdntroubleshoot.sdn_dump:UNKNOWN packet 66:9c:7f
    :16:8e:38
71 INFO:sdntroubleshoot.sdn_dump:UNKNOWN packet 66:9c:7f
    :16:8e:38
72 INFO:sdntroubleshoot.sdn_dump:UNKNOWN packet 6a:24:fd:
    a5:16:0d
73 INFO:sdntroubleshoot.sdn_dump:UNKNOWN packet 6a:24:fd:
    a5:16:0d
74 INFO:sdntroubleshoot.sdn_dump:UNKNOWN packet 8e:c8:2d
    :40:5e:b6
75 INFO:sdntroubleshoot.sdn_dump:UNKNOWN packet 72:75:cc:
    e8:56:76
76 INFO:sdntroubleshoot.sdn_dump:UNKNOWN packet 8e:c8:2d
    :40:5e:b6
```

```
77 INFO:sdntroubleshoot.sdn_dump:UNKNOWN packet 72:75:cc:
       e8:56:76
78 INFO:sdntroubleshoot.sdn_dump:UNKNOWN packet 8e:c8:2d
       :40:5e:b6
79 INFO:sdntroubleshoot.sdn_dump:UNKNOWN packet 8e:c8:2d
       :40:5e:b6
80 INFO:sdntroubleshoot.sdn_dump:UNKNOWN packet 8e:c8:2d
       :40:5e:b6
81 INFO:sdntroubleshoot.sdn_dump:UNKNOWN packet 72:75:cc:
       e8:56:76
82 INFO:sdntroubleshoot.sdn_dump:UNKNOWN packet 8e:c8:2d
       :40:5e:b6
83 INFO:sdntroubleshoot.sdn_dump:UNKNOWN packet 8e:c8:2d
       :40:5e:b6
84 INFO:sdntroubleshoot.sdn_dump:UNKNOWN packet 8e:c8:2d
       :40:5e:b6
85 INFO:sdntroubleshoot.sdn_dump:UNKNOWN packet 8e:c8:2d
       :40:5e:b6
86 INFO:openflow.discovery:link detected:
       00-00-00-00-00-01.2 -> 00-00-00-00-00-02.2
87 INFO:openflow.spanning_tree:5 ports changed
88 INFO:sdntroubleshoot.sdn_dump:UNKNOWN packet 5e:9a
       :34:1e:c6:b8
89 INFO:sdntroubleshoot.sdn_dump:UNKNOWN packet 5e:9a
       :34:1e:c6:b8
90 INFO:sdntroubleshoot.sdn_dump:UNKNOWN packet 5e:9a
       :34:1e:c6:b8
91 INFO:sdntroubleshoot.sdn_dump:UNKNOWN packet 5e:9a
       :34:1e:c6:b8
92 INFO:sdntroubleshoot.sdn_dump:UNKNOWN packet 5e:9a
       :34:1e:c6:b8
93 INFO:sdntroubleshoot.sdn_dump:UNKNOWN packet 5e:9a
       :34:1e:c6:b8
94 INFO:sdntroubleshoot.sdn_dump:UNKNOWN packet 5e:9a
       :34:1e:c6:b8
95 INFO:sdntroubleshoot.sdn_dump:UNKNOWN packet 5e:9a
       :34:1e:c6:b8
96 INFO:sdntroubleshoot.sdn_dump:UNKNOWN packet 5e:9a
       :34:1e:c6:b8
97 INFO:openflow.discovery:link detected:
       00-00-00-00-00-05.3 -> 00-00-00-00-00-06.2
98 INFO:openflow.discovery:link detected:
       00-00-00-00-00-05.2 -> 00-00-00-00-00-04.3
99 INFO:openflow.discovery:link detected:
       00-00-00-00-00-03.3 -> 00-00-00-00-00-04.2
100 INFO:openflow.discovery:link detected:
       00-00-00-00-00-03.2 -> 00-00-00-00-00-02.3
101 INFO:openflow.spanning_tree:4 ports changed
```

```
102 INFO:openflow.discovery:link detected:
        00-00-00-00-00-04.3 -> 00-00-00-00-00-05.2
103 INFO:openflow.spanning_tree:6 ports changed
104 INFO:openflow.discovery:link detected:
        00-00-00-00-00-04.2 -> 00-00-00-00-00-03.3
105 INFO:openflow.discovery:link detected:
        00-00-00-00-00-07.3 -> 00-00-00-00-00-08.2
106 INFO:openflow.spanning_tree:6 ports changed
107 INFO:openflow.discovery:link detected:
        00-00-00-00-00-07.2 -> 00-00-00-00-00-06.3
108 INFO:openflow.spanning_tree:1 ports changed
109 INFO:openflow.discovery:link detected:
        00-00-00-00-00-06.3 -> 00-00-00-00-00-07.2
110 INFO:openflow.spanning_tree:4 ports changed
111 INFO:openflow.discovery:link detected:
        00-00-00-00-00-06.2 -> 00-00-00-00-00-05.3
112 INFO:openflow.discovery:link detected:
        00-00-00-00-00-0a.2 -> 00-00-00-00-00-09.3
113 INFO:openflow.spanning_tree:5 ports changed
114 Sun 14:53:37 DevID: 00-00-00-00-00-01 ARP: pkt b6
        :97:29:5e:02:8f > ff:ff:ff:ff:ff:ff, hw b6:97:29:5e
        :02:8f > 00:00:00:00:00:00: 1:REQUEST
115 Sun 14:53:37 DevID: 00-00-00-00-00-02 ARP: pkt b6
        :97:29:5e:02:8f > ff:ff:ff:ff:ff:ff, hw b6:97:29:5e
        :02:8f > 00:00:00:00:00:00: 1:REQUEST
116 Sun 14:53:37 DevID: 00-00-00-00-00-03 ARP: pkt b6
        :97:29:5e:02:8f > ff:ff:ff:ff:ff:ff, hw b6:97:29:5e
        :02:8f > 00:00:00:00:00:00: 1:REQUEST
117 Sun 14:53:37 DevID: 00-00-00-00-00-04 ARP: pkt b6
        :97:29:5e:02:8f > ff:ff:ff:ff:ff:ff, hw b6:97:29:5e
        :02:8f > 00:00:00:00:00:00: 1:REQUEST
118 Sun 14:53:37 DevID: 00-00-00-00-00-05 ARP: pkt b6
        :97:29:5e:02:8f > ff:ff:ff:ff:ff:ff, hw b6:97:29:5e
        :02:8f > 00:00:00:00:00:00: 1:REQUEST
119 Sun 14:53:37 DevID: 00-00-00-00-00-06 ARP: pkt b6
        :97:29:5e:02:8f > ff:ff:ff:ff:ff:ff, hw b6:97:29:5e
        :02:8f > 00:00:00:00:00:00: 1:REQUEST
120 Sun 14:53:37 DevID: 00-00-00-00-00-07 ARP: pkt b6
        :97:29:5e:02:8f > ff:ff:ff:ff:ff:ff, hw b6:97:29:5e
        :02:8f > 00:00:00:00:00:00: 1:REQUEST
121 Sun 14:53:37 DevID: 00-00-00-00-00-08 ARP: pkt b6
        :97:29:5e:02:8f > ff:ff:ff:ff:ff:ff, hw b6:97:29:5e
        :02:8f > 00:00:00:00:00:00: 1:REQUEST
122 Sun 14:53:37 DevID: 00-00-00-00-00-09 ARP: pkt b6
        :97:29:5e:02:8f > ff:ff:ff:ff:ff:ff, hw b6:97:29:5e
        :02:8f > 00:00:00:00:00:00: 1:REQUEST
123 Sun 14:53:37 DevID: 00-00-00-00-00-0a ARP: pkt b6
        :97:29:5e:02:8f > ff:ff:ff:ff:ff:ff, hw b6:97:29:5e
```

```
    :02:8f > 00:00:00:00:00:00: 1:REQUEST
124 Sun 14:53:37 DevID: 00-00-00-00-00-0a ARP: pkt 72:75:
    cc:e8:56:76 > b6:97:29:5e:02:8f, hw 72:75:cc:e8
    :56:76 > b6:97:29:5e:02:8f: 2:REPLY
125 Sun 14:53:37 DevID: 00-00-00-00-00-09 ARP: pkt 72:75:
    cc:e8:56:76 > b6:97:29:5e:02:8f, hw 72:75:cc:e8
    :56:76 > b6:97:29:5e:02:8f: 2:REPLY
126 Sun 14:53:37 DevID: 00-00-00-00-00-08 ARP: pkt 72:75:
    cc:e8:56:76 > b6:97:29:5e:02:8f, hw 72:75:cc:e8
    :56:76 > b6:97:29:5e:02:8f: 2:REPLY
127 Sun 14:53:37 DevID: 00-00-00-00-00-07 ARP: pkt 72:75:
    cc:e8:56:76 > b6:97:29:5e:02:8f, hw 72:75:cc:e8
    :56:76 > b6:97:29:5e:02:8f: 2:REPLY
128 Sun 14:53:37 DevID: 00-00-00-00-00-06 ARP: pkt 72:75:
    cc:e8:56:76 > b6:97:29:5e:02:8f, hw 72:75:cc:e8
    :56:76 > b6:97:29:5e:02:8f: 2:REPLY
129 Sun 14:53:37 DevID: 00-00-00-00-00-05 ARP: pkt 72:75:
    cc:e8:56:76 > b6:97:29:5e:02:8f, hw 72:75:cc:e8
    :56:76 > b6:97:29:5e:02:8f: 2:REPLY
130 Sun 14:53:37 DevID: 00-00-00-00-00-04 ARP: pkt 72:75:
    cc:e8:56:76 > b6:97:29:5e:02:8f, hw 72:75:cc:e8
    :56:76 > b6:97:29:5e:02:8f: 2:REPLY
131 Sun 14:53:37 DevID: 00-00-00-00-00-03 ARP: pkt 72:75:
    cc:e8:56:76 > b6:97:29:5e:02:8f, hw 72:75:cc:e8
    :56:76 > b6:97:29:5e:02:8f: 2:REPLY
132 Sun 14:53:37 DevID: 00-00-00-00-00-02 ARP: pkt 72:75:
    cc:e8:56:76 > b6:97:29:5e:02:8f, hw 72:75:cc:e8
    :56:76 > b6:97:29:5e:02:8f: 2:REPLY
133 Sun 14:53:37 DevID: 00-00-00-00-00-01 ARP: pkt 72:75:
    cc:e8:56:76 > b6:97:29:5e:02:8f, hw 72:75:cc:e8
    :56:76 > b6:97:29:5e:02:8f: 2:REPLY
134 Sun 14:53:37 DevID: 00-00-00-00-00-01 ICMP: pkt b6
    :97:29:5e:02:8f > 72:75:cc:e8:56:76, ip 10.0.0.1 >
    10.0.0.10: 8:ECHO_REQUEST
135 Sun 14:53:37 DevID: 00-00-00-00-00-02 ICMP: pkt b6
    :97:29:5e:02:8f > 72:75:cc:e8:56:76, ip 10.0.0.1 >
    10.0.0.10: 8:ECHO_REQUEST
136 Sun 14:53:37 DevID: 00-00-00-00-00-03 ICMP: pkt b6
    :97:29:5e:02:8f > 72:75:cc:e8:56:76, ip 10.0.0.1 >
    10.0.0.10: 8:ECHO_REQUEST
137 Sun 14:53:37 DevID: 00-00-00-00-00-04 ICMP: pkt b6
    :97:29:5e:02:8f > 72:75:cc:e8:56:76, ip 10.0.0.1 >
    10.0.0.10: 8:ECHO_REQUEST
138 Sun 14:53:37 DevID: 00-00-00-00-00-05 ICMP: pkt b6
    :97:29:5e:02:8f > 72:75:cc:e8:56:76, ip 10.0.0.1 >
    10.0.0.10: 8:ECHO_REQUEST
139 Sun 14:53:37 DevID: 00-00-00-00-00-06 ICMP: pkt b6
    :97:29:5e:02:8f > 72:75:cc:e8:56:76, ip 10.0.0.1 >
```

```
        10.0.0.10: 8:ECHO_REQUEST
140 Sun 14:53:37 DevID: 00-00-00-00-00-07 ICMP: pkt b6
        :97:29:5e:02:8f > 72:75:cc:e8:56:76, ip 10.0.0.1 >
        10.0.0.10: 8:ECHO_REQUEST
141 Sun 14:53:38 DevID: 00-00-00-00-00-08 ICMP: pkt b6
        :97:29:5e:02:8f > 72:75:cc:e8:56:76, ip 10.0.0.1 >
        10.0.0.10: 8:ECHO_REQUEST
142 Sun 14:53:38 DevID: 00-00-00-00-00-09 ICMP: pkt b6
        :97:29:5e:02:8f > 72:75:cc:e8:56:76, ip 10.0.0.1 >
        10.0.0.10: 8:ECHO_REQUEST
143 Sun 14:53:38 DevID: 00-00-00-00-00-0a ICMP: pkt b6
        :97:29:5e:02:8f > 72:75:cc:e8:56:76, ip 10.0.0.1 >
        10.0.0.10: 8:ECHO_REQUEST
144 Sun 14:53:38 DevID: 00-00-00-00-00-0a ICMP: pkt 72:75:
        cc:e8:56:76 > b6:97:29:5e:02:8f, ip 10.0.0.10 >
        10.0.0.1: 0:ECHO_REPY
145 Sun 14:53:38 DevID: 00-00-00-00-00-09 ICMP: pkt 72:75:
        cc:e8:56:76 > b6:97:29:5e:02:8f, ip 10.0.0.10 >
        10.0.0.1: 0:ECHO_REPY
146 Sun 14:53:38 DevID: 00-00-00-00-00-08 ICMP: pkt 72:75:
        cc:e8:56:76 > b6:97:29:5e:02:8f, ip 10.0.0.10 >
        10.0.0.1: 0:ECHO_REPY
147 Sun 14:53:38 DevID: 00-00-00-00-00-07 ICMP: pkt 72:75:
        cc:e8:56:76 > b6:97:29:5e:02:8f, ip 10.0.0.10 >
        10.0.0.1: 0:ECHO_REPY
148 Sun 14:53:38 DevID: 00-00-00-00-00-06 ICMP: pkt 72:75:
        cc:e8:56:76 > b6:97:29:5e:02:8f, ip 10.0.0.10 >
        10.0.0.1: 0:ECHO_REPY
149 Sun 14:53:38 DevID: 00-00-00-00-00-05 ICMP: pkt 72:75:
        cc:e8:56:76 > b6:97:29:5e:02:8f, ip 10.0.0.10 >
        10.0.0.1: 0:ECHO_REPY
150 Sun 14:53:38 DevID: 00-00-00-00-00-04 ICMP: pkt 72:75:
        cc:e8:56:76 > b6:97:29:5e:02:8f, ip 10.0.0.10 >
        10.0.0.1: 0:ECHO_REPY
151 Sun 14:53:38 DevID: 00-00-00-00-00-03 ICMP: pkt 72:75:
        cc:e8:56:76 > b6:97:29:5e:02:8f, ip 10.0.0.10 >
        10.0.0.1: 0:ECHO_REPY
152 Sun 14:53:38 DevID: 00-00-00-00-00-02 ICMP: pkt 72:75:
        cc:e8:56:76 > b6:97:29:5e:02:8f, ip 10.0.0.10 >
        10.0.0.1: 0:ECHO_REPY
153 Sun 14:53:38 DevID: 00-00-00-00-00-01 ICMP: pkt 72:75:
        cc:e8:56:76 > b6:97:29:5e:02:8f, ip 10.0.0.10 >
        10.0.0.1: 0:ECHO_REPY
154 Sun 14:53:43 DevID: 00-00-00-00-00-0a ARP: pkt 72:75:
        cc:e8:56:76 > b6:97:29:5e:02:8f, hw 72:75:cc:e8
        :56:76 > 00:00:00:00:00:00: 1:REQUEST
155 Sun 14:53:43 DevID: 00-00-00-00-00-09 ARP: pkt 72:75:
        cc:e8:56:76 > b6:97:29:5e:02:8f, hw 72:75:cc:e8
```

```
    :56:76 > 00:00:00:00:00:00: 1:REQUEST
156 Sun 14:53:43 DevID: 00-00-00-00-00-08 ARP: pkt 72:75:
    cc:e8:56:76 > b6:97:29:5e:02:8f, hw 72:75:cc:e8
    :56:76 > 00:00:00:00:00:00: 1:REQUEST
157 Sun 14:53:43 DevID: 00-00-00-00-00-07 ARP: pkt 72:75:
    cc:e8:56:76 > b6:97:29:5e:02:8f, hw 72:75:cc:e8
    :56:76 > 00:00:00:00:00:00: 1:REQUEST
158 Sun 14:53:43 DevID: 00-00-00-00-00-06 ARP: pkt 72:75:
    cc:e8:56:76 > b6:97:29:5e:02:8f, hw 72:75:cc:e8
    :56:76 > 00:00:00:00:00:00: 1:REQUEST
159 Sun 14:53:43 DevID: 00-00-00-00-00-05 ARP: pkt 72:75:
    cc:e8:56:76 > b6:97:29:5e:02:8f, hw 72:75:cc:e8
    :56:76 > 00:00:00:00:00:00: 1:REQUEST
160 Sun 14:53:43 DevID: 00-00-00-00-00-04 ARP: pkt 72:75:
    cc:e8:56:76 > b6:97:29:5e:02:8f, hw 72:75:cc:e8
    :56:76 > 00:00:00:00:00:00: 1:REQUEST
161 Sun 14:53:43 DevID: 00-00-00-00-00-03 ARP: pkt 72:75:
    cc:e8:56:76 > b6:97:29:5e:02:8f, hw 72:75:cc:e8
    :56:76 > 00:00:00:00:00:00: 1:REQUEST
162 Sun 14:53:43 DevID: 00-00-00-00-00-02 ARP: pkt 72:75:
    cc:e8:56:76 > b6:97:29:5e:02:8f, hw 72:75:cc:e8
    :56:76 > 00:00:00:00:00:00: 1:REQUEST
163 Sun 14:53:43 DevID: 00-00-00-00-00-01 ARP: pkt 72:75:
    cc:e8:56:76 > b6:97:29:5e:02:8f, hw 72:75:cc:e8
    :56:76 > 00:00:00:00:00:00: 1:REQUEST
164 Sun 14:53:43 DevID: 00-00-00-00-00-01 ARP: pkt b6
    :97:29:5e:02:8f > 72:75:cc:e8:56:76, hw b6:97:29:5e
    :02:8f > 72:75:cc:e8:56:76: 2:REPLY
165 Sun 14:53:43 DevID: 00-00-00-00-00-02 ARP: pkt b6
    :97:29:5e:02:8f > 72:75:cc:e8:56:76, hw b6:97:29:5e
    :02:8f > 72:75:cc:e8:56:76: 2:REPLY
166 Sun 14:53:43 DevID: 00-00-00-00-00-03 ARP: pkt b6
    :97:29:5e:02:8f > 72:75:cc:e8:56:76, hw b6:97:29:5e
    :02:8f > 72:75:cc:e8:56:76: 2:REPLY
167 Sun 14:53:43 DevID: 00-00-00-00-00-04 ARP: pkt b6
    :97:29:5e:02:8f > 72:75:cc:e8:56:76, hw b6:97:29:5e
    :02:8f > 72:75:cc:e8:56:76: 2:REPLY
168 Sun 14:53:43 DevID: 00-00-00-00-00-05 ARP: pkt b6
    :97:29:5e:02:8f > 72:75:cc:e8:56:76, hw b6:97:29:5e
    :02:8f > 72:75:cc:e8:56:76: 2:REPLY
169 Sun 14:53:43 DevID: 00-00-00-00-00-06 ARP: pkt b6
    :97:29:5e:02:8f > 72:75:cc:e8:56:76, hw b6:97:29:5e
    :02:8f > 72:75:cc:e8:56:76: 2:REPLY
170 Sun 14:53:43 DevID: 00-00-00-00-00-07 ARP: pkt b6
    :97:29:5e:02:8f > 72:75:cc:e8:56:76, hw b6:97:29:5e
    :02:8f > 72:75:cc:e8:56:76: 2:REPLY
171 Sun 14:53:43 DevID: 00-00-00-00-00-08 ARP: pkt b6
    :97:29:5e:02:8f > 72:75:cc:e8:56:76, hw b6:97:29:5e
```

```
    :02:8f > 72:75:cc:e8:56:76: 2:REPLY
172 Sun 14:53:43 DevID: 00-00-00-00-00-09 ARP: pkt b6
    :97:29:5e:02:8f > 72:75:cc:e8:56:76, hw b6:97:29:5e
    :02:8f > 72:75:cc:e8:56:76: 2:REPLY
173 Sun 14:53:43 DevID: 00-00-00-00-00-0a ARP: pkt b6
    :97:29:5e:02:8f > 72:75:cc:e8:56:76, hw b6:97:29:5e
    :02:8f > 72:75:cc:e8:56:76: 2:REPLY
174 ^CINFO:core:Going down...
175 INFO:openflow.of_01:[00-00-00-00-00-01 5] disconnected
176 INFO:openflow.of_01:[00-00-00-00-00-02 2] disconnected
177 INFO:openflow.of_01:[00-00-00-00-00-03 6] disconnected
178 INFO:openflow.of_01:[00-00-00-00-00-04 7] disconnected
179 INFO:openflow.of_01:[00-00-00-00-00-05 1] disconnected
180 INFO:openflow.of_01:[00-00-00-00-00-06 9] disconnected
181 INFO:openflow.of_01:[00-00-00-00-00-07 8] disconnected
182 INFO:openflow.of_01:[00-00-00-00-00-08 3] disconnected
183 INFO:openflow.of_01:[00-00-00-00-00-09 4] disconnected
184 INFO:openflow.of_01:[00-00-00-00-00-0a 10]
    disconnected
185 INFO:core:Down.
```

Appendix C

Sdn_dump Source Code

C.1 Sdn_dump.py

```
1  ########################################
2  # Network wide tcpdump
3  # @Author Haris Sistek
4  ########################################
5
6  from pox.core import core
7  import pox.openflow.libopenflow_01 as of
8  from pox.lib.util import dpid_to_str
9  import pox.lib.packet as packet
10 import datetime, time
11 from pox.openflow.of_json import *
12 import re
13 import violation_checker as vc # see file
       violation_checker.py
14
15 # spanning tree protocol from pox
16 import pox.openflow.discovery as discov
17 import pox.openflow.spanning_tree as spanning_tree
18
19 log = core.getLogger()
20
21 # Layer 1 Vars:
22 checker = None # violation_checker class var, declared
       in launch()
23 layer1_correct = True # Helps us decide what layer we
       should search on
24 last_packet = None # The packet that triggered the
       violation
25
26 # Port stat vars:
27 count = 0
28 dev_port_ip = {}
```

```
29 known_maps = []
30
31 # Remember number of bytes sent:
32 ip_bytes_sent = {}
33 ip_packets_sent = {}
34 ip_bytes_recv = {}
35 ip_packets_recv = {}
36
37 # Layer 2 Vars:
38 yet_to_do = True
39 switch_count = 0
40
41 '''
42 Will␣map␣IP␣to␣port/dev␣relationship
43 '''
44 def add_host(dev,port,ip):
45     global dev_port_ip
46     global known_maps
47     if ip in known_maps:
48         return
49
50     if dev_port_ip.get(dev):
51         info = dev_port_ip.get(dev)
52         info[ip] = port
53         info[port] = ip
54         dev_port_ip[dev] = info
55         known_maps.append(ip) # just map once
56         return
57
58     info = {}
59     info[ip] = port
60     info[port] = ip
61     dev_port_ip[dev] = info
62     known_maps.append(ip) # just map once
63
64 '''
65 Update␣lates␣port␣stats␣given␣from␣network␣devices,␣
    map␣to␣IP
66 '''
67 def add_port_entry(ip, bsent, brecv, psent, precv):
68     global ip_bytes_sent
69     global ip_bytes_recv
70     global ip_packets_sent
71     global ip_packets_recv
72
73     ip_bytes_sent[ip] = bsent
74     ip_bytes_recv[ip] = brecv
75     ip_packets_sent[ip] = psent
```

```
76      ip_packets_recv[ip] = precv
77
78
79  def handle_port_stats(event):
80      stats = event.stats
81      for stat in stats:
82        if dev_port_ip.get(dpid_to_str(event.dpid)):# if
              entry exists
83          if dev_port_ip.get(dpid_to_str(event.dpid)).
              get(stat.port_no):# if entry exists
84            add_port_entry(dev_port_ip[dpid_to_str(event
                .dpid)][stat.port_no], stat.tx_bytes,
85                           stat.rx_bytes, stat.
                               tx_packets, stat.
                               rx_packets)
86      if layer1_correct:
87          if checker:
88              violation = checker.check_if_ports_legal(
                    ip_bytes_sent,
89                                                  ip_bytes_recv
                                                        ,
                                                        ip_packets_sent
                                                        ,
                                                        ip_packets_recv
                                                        )
90              decider(violation, None)
91
92  def handle_flow_requests(event):
93      stats = flow_stats_to_list(event.stats)
94      ips = None
95      ports = None
96      if last_packet:
97          match = re.search(r'ip\s+(?P<ip_src>.+)\s+>\s
                +(?P<ip_dst>.+):\s+', last_packet)
98          if match:
99              ips = match.groupdict()
100
101         match2 = re.search(r'srcp\s+(?P<sport>\d+)\s+
                dstp\s+(?P<dport>\d+)', last_packet)
102         if match2:
103             ports = match2.groupdict()
104
105
106     print "## Flow Table for Deivce:", event.dpid, "##
            "
107     entry = 1
108     for stat in stats:
109         print "> Entry %d Match:" % entry
```

```
110             print stat["match"]
111             print ">> Entry Action"
112             print stat["actions"]
113             print ">> Byte Count"
114             print stat["byte_count"]
115             entry = entry + 1
116
117
118 def check_switch_connectivity():
119     count = 0
120     print "Checking switch connectivity:"
121     for con in core.openflow._connections.values():
122         print "Switch", con, "is alive"
123         count = count + 1
124     print switch_count - count, "device(s) unaccounted
        for"
125     return count == switch_count
126
127 def start_spanning_tree():
128     global yet_to_do
129     if yet_to_do:
130         print "Searching for forwarding loops with
            spanning tree..."
131         #pox.openflow.discovery
132         discov.launch()
133         #pox.openflow.spanning_tree --no-flood --hold-
            down
134         spanning_tree.launch()
135         yet_to_do = False
136
137 # Get timestamp in format of HH:MM:SS = 23:13:20
138 def timestamp():
139   stamp = time.time() # see if i have to set the
        locale on the time so that it doesnt confuese
        norwegian and english
140   return datetime.datetime.fromtimestamp(stamp).
        strftime('%a %H:%M:%S')
141
142 def decider(bool_val, packet):
143   global layer1_correct
144   global last_packet
145
146   if bool_val:
147     last_packet = packet
148     print "### Stage 2: Started ###"
149     print "Question A: No, Policy does not match
          actual behaviour"
```

```
150       print "Searching␣Question␣B:␣'Does␣policy␣match␣
              device␣state?'"
151       layer1_correct = False
152       # Seatch deeper on device state (Layer 2)
153       #start_spanning_tree()
154       con_check = check_switch_connectivity()
155
156       # Request flow stats (Layer 3 Question D)
157       if con_check: # if Yes on question B
158           print "###␣Stage␣3:␣Started␣###"
159           print "Question␣B:␣Yes,␣Device␣state␣matches␣
                  Policy."
160           print "Searching␣␣Question␣D:␣'Does␣device␣
                  state␣match␣hardware?'"
161           for con in core.openflow._connections.values()
                  :
162               print "Requesting␣flow␣table␣entries␣from␣
                      device", con
163               con.send(of.ofp_stats_request(body=of.
                      ofp_flow_stats_request()))
164       else:
165           print "###␣Stage␣3:␣Started"
166           print "Question␣B:␣No,␣Device␣state␣doesn't␣
                  match␣Policy."
167           print "Searching␣␣Question␣C:␣'Does␣physical␣
                  view␣␣match␣Device␣State?'"
168
169
170 ############################
171 # Different packet handling:
172 ############################
173 def handle_arp(dev_id,packet):
174   arp_packet = packet.payload
175   hwsrc = arp_packet.hwsrc
176   hwdst = arp_packet.hwdst
177   opcode = arp_packet.opcode
178   if opcode == 1:
179     opcode = "1:REQUEST"
180   elif opcode == 2:
181     opcode = "2:REPLY"
182   elif opcode == 3:
183     opcode = "3:REV_REQUEST"
184   elif opcode == 4:
185     opcode = "4:REV_REPLY"
186   else:
187     opcode = "x:UNSET"
188   #log.info("%s DevID: %s ARP: pkt %s > %s, hw %s > %s
          : %s", timestamp(), dev_id, packet.src, packet.
```

```
        dst, hwsrc, hwdst, opcode)
189     x = "%s DevID: %s ARP: pkt %s > %s, hw %s > %s: %s"
          % (timestamp(), dev_id, packet.src, packet.dst,
        hwsrc, hwdst, opcode)
190     if layer1_correct:
191       print x
192       if checker:
193           checker.check_if_legal(x)
194
195 ########################
196 # Handle the ip packets
197 ########################
198
199 def handle_icmp(dev_id, packet, ip_packet, srcip,
      dstip):
200     icmp_packet = ip_packet.payload
201     ping_type = icmp_packet.type
202     ping_code = icmp_packet.code
203     ping_csum = icmp_packet.csum
204     if ping_type == 0:
205       ping_type = "0:ECHO_REPY"
206     elif ping_type == 3:
207       ping_type = "3:DEST_UNREACH"
208     elif ping_type == 4:
209       ping_type = "4:SRC_QUENCH"
210     elif ping_type == 5:
211       ping_type = "5:REDIRECT"
212     elif ping_type == 8:
213       ping_type = "8:ECHO_REQUEST"
214     elif ping_type == 11:
215       ping_type = "11:TIME_EXCEED"
216     else:
217       pass
218     #log.info("%s DevID: %s ICMP: pkt %s > %s, ip %s > %
          s: %s", timestamp(), dev_id, packet.src, packet.
          dst, srcip, dstip, ping_type)
219     x = "%s DevID: %s ICMP: pkt %s > %s, ip %s > %s: %s
          " % (timestamp(), dev_id, packet.src, packet.dst,
           srcip, dstip, ping_type)
220     if layer1_correct:
221       print x
222       if checker:
223           violation = checker.check_if_legal(x)
224           decider(violation, x)
225
226 def handle_tcp(dev_id, packet, ip_packet, srcip, dstip
      ):
227     tcp = ip_packet.payload
```

```
228   srcport = tcp.srcport
229   dstport = tcp.dstport
230   seq = tcp.seq
231   ack = tcp.ack
232   flags = tcp.flags
233   x =  "%s DevID: %s TCP: pkt %s > %s, ip %s > %s:
          srcp %s dstp %s, seq %s, ack %s, flags %s" % (
          timestamp(), dev_id, packet.src, packet.dst,
          srcip, dstip, srcport, dstport, seq, ack, flags)
234   if layer1_correct:
235     print x
236     if checker:
237         violation = checker.check_if_legal(x)
238         decider(violation, x)
239
240 def handle_udp(dev_id, packet, ip_packet, srcip, dstip
      ):
241   udp  = ip_packet.payload
242   srcport = udp.srcport
243   dstport = udp.dstport
244   x =  "%s DevID: %s UDP: pkt %s > %s, ip %s > %s:
          srcp %s dstp %s" % (timestamp(), dev_id, packet.
          src, packet.dst, srcip, dstip, srcport, dstport)
245   if layer1_correct:
246     print x
247     if checker:
248         violation = checker.check_if_legal(x)
249         decider(violation, x)
250
251 def handle_ip(dev_id,port,packet):
252   ip_packet = packet.payload
253   srcip = ip_packet.srcip
254   dstip = ip_packet.dstip
255
256   add_host(dev_id, port, srcip)
257
258   if ip_packet.find("icmp"):
259     handle_icmp(dev_id, packet, ip_packet, srcip,
          dstip)
260   elif ip_packet.find("tcp"):
261     handle_tcp(dev_id, packet, ip_packet, srcip, dstip
          )
262   elif ip_packet.find("udp"):
263     handle_udp(dev_id, packet, ip_packet, srcip, dstip
          )
264
265
266 def _handle_PacketIn(event):
```

```
267     global count
268     packet = event.parsed
269     if packet.find("arp"):
270       handle_arp(dpid_to_str(event.dpid),packet)
271     elif packet.find("ipv4"):
272       handle_ip(dpid_to_str(event.dpid),event.port,
            packet)
273     else:
274       log.info("UNKNOWN packet %s", packet.src)
275
276     count = count + 1
277     if count == 5: # every 5 packets recevied check data
          stats/small number for testing
278       send_requests()
279       count  = 0
280
281 def send_requests():
282       for con in core.openflow._connections.values():
283           con.send(of.ofp_stats_request(body=of.
              ofp_port_stats_request()))
284
285 def launch (switch = "", mode= ""):
286     global checker
287     global switch_count
288     print "Number of switches expected:", switch
289     switch_count = int(switch)
290     start_spanning_tree()
291     if "2" in mode or "3" in mode:
292         checker = vc.Violation_Checker(switch, mode)
293     core.openflow.addListenerByName("PacketIn",
          _handle_PacketIn)
294     core.openflow.addListenerByName("PortStatsReceived",
          handle_port_stats)
295     core.openflow.addListenerByName("FlowStatsReceived",
          handle_flow_requests)
296     print "### Stage 1: Started ###"
297     print "Searching Question A: 'Does policy match
          actual behaviour?'"
```

C.2 Violation_checker.py

```
1 # Checks if packets violate the network policy
2 # @ Author Haris Sistek
3 import os
4 import re
5 import time, datetime
6 import automator as auto
7
```

```
8  class Violation_Checker(object):
9    '''
10 ␣␣Interpret␣all␣possible␣options␣given␣by␣a␣rule␣add␣
       the␣to␣a␣dict
11 ␣␣Returns␣a␣dict␣with␣option␣variables,␣or␣None␣if␣no␣
       options␣are␣found
12 ␣␣'''
13   def interpret_options(self, rule):
14     match = re.search(r'sport\s+(?P<sport>.+)\s+dport\
         s+(?P<dport>.+)', rule)
15     if match:
16       return match.groupdict()
17     else:
18       just_sport_match = re.search(r'sport\s+(?P<sport
           >.+)', rule)
19       if just_sport_match:
20         return just_sport_match.groupdict()
21       else:
22         just_dport_match = re.search(r'dport\s+(?P<
             dport>.+)', rule)
23       if just_dport_match:
24         return just_dport_match.groupdict()
25       else:
26         return None
27
28   '''
29 ␣␣Use␣to␣merge␣the␣rule␣dicts␣that␣are␣extracted␣from␣
       the␣rules.
30 ␣␣prot␣=␣dict␣with␣the␣protocol␣name
31 ␣␣type_dict␣=␣values␣extracted␣from␣Time,␣Date␣or␣Vlan
       ␣rules
32 ␣␣options␣=␣values␣extracted␣from␣"interpret_options"
33 ␣␣'''
34   def merge_dicts(self,prot,type_dict,options):
35     ret_dict = {}
36     if options:
37       ret_dict = dict(list(prot.items()) + list(
           type_dict.items()) + list(options.items()))
38     else:
39       ret_dict = dict(list(prot.items()) + list(
           type_dict.items()))
40     return ret_dict
41
42
43   '''
44 ␣␣Extract␣all␣data␣needed␣to␣interpet␣"Data"␣rules,␣
       returns␣dict␣of␣the␣data␣stored
45 ␣␣'''
```

```
46    def interpret_data_rule(self,prot,rule):
47      ret_dict = {}
48      ret_dict["rule_type"] = "Data"
49      ret_dict["rule_string"] = rule
50      ret_dict["lim"] = rule # just simple search later,
            easier the to split up the line atm
51      match = re.search(r'^Data\s+(?P<lim>.+)\s+(?P<
            notation>.+)\s+from\s+(?P<from>.+)',rule)
52      if match:
53        ret_dict = dict(list(ret_dict.items()) + list(
              match.groupdict().items()))
54        return self.merge_dicts(prot, ret_dict, {})
55      else:
56        match = re.search(r'^Data\s+(?P<lim>.+)\s+(?P<
              notation>.+)\s+to\s+(?P<to>.+)',rule)
57        if match:
58          ret_dict = dict(list(ret_dict.items()) + list(
                match.groupdict().items()))
59          return self.merge_dicts(prot, ret_dict, {})
60        else:
61          return None
62
63    '''
64  ␣␣Helps␣us␣convert␣the␣policy␣data␣size␣notation␣into␣
      bytes,␣because␣this␣is␣what
65  ␣␣openflow␣port␣stat␣request␣is␣returning
66  ␣␣'''
67    def convert_notation_to_bytes(self, lim, notation):
68      notation = notation.upper()
69      if notation == "B":
70          return int(lim)
71      elif notation == "KB":
72          return int(lim) * 1024
73      elif notation == "MB":
74          return int(lim) * 1024 * 1024
75      elif notation == "GB":
76          return int(lim) * 1024 * 1024 * 1024
77      elif notation == "TB":
78          return int(lim) * 1024 * 1024 * 1024 * 1024
79      else:
80          return 0
81
82    '''
83  ␣␣Extract␣all␣the␣data␣needed␣from␣"Time"␣rule,␣return
      ␣a␣dict␣of␣that␣data
84  ␣␣'''
85    def interpret_time_rule(self,prot,rule):
86      ret_dict = {}
```

```
87      ret_dict["rule_type"] = "Time"
88      ret_dict["rule_string"] = rule
89      match = re.search(r'^Time\s+(?P<start_time>.+)\s+
           to\s+(?P<end_time>.+)\s+block\s+(?P<from>.+)\s+
           to\s+(?P<to>\S+)\s+', rule)
90      if match:
91        print "match1"
92        options = self.interpret_options(rule)
93        ret_dict = dict(list(ret_dict.items()) + list(
             match.groupdict().items()))
94        return self.merge_dicts(prot, ret_dict, options)
95      else:
96        match = re.search(r'^Time\s+(?P<start_time>.+)\s
             +to\s+(?P<end_time>.+)\s+block\s+(?P<from>\S
             +)\s+', rule)
97        if match:
98          options = self.interpret_options(rule)
99          ret_dict = dict(list(ret_dict.items()) + list(
               match.groupdict().items()))
100         return self.merge_dicts(prot, ret_dict,
               options)
101       else:
102         return None
103
104   '''
105 ␣␣Extract␣all␣the␣data␣from␣"Date"␣rule,␣return␣a␣dict
      ␣of␣that␣data
106 ␣␣'''
107   def interpret_date_rule(self,prot,rule):
108     ret_dict = {}
109     ret_dict["rule_type"] = "Date"
110     ret_dict["rule_string"] = rule
111     match = re.search(r'^Date\s+(?P<start_date>.+)\s+
           to\s+(?P<end_date>.+)\s+block\s+(?P<from>.+)\s+
           to\s+(?P<to>\S+)\s+', rule)
112     if match:
113       options = self.interpret_options(rule)
114       ret_dict = dict(list(ret_dict.items()) + list(
             match.groupdict().items()))
115       return self.merge_dicts(prot, ret_dict, options)
116     else:
117       match = re.search(r'^Date\s+(?P<start_date>.+)\s
             +to\s+(?P<end_date>.+)\s+block\s+(?P<from>\S
             +)', rule)
118       if match:
119         options = self.interpret_options(rule)
120         ret_dict = dict(list(ret_dict.items()) + list(
               match.groupdict().items()))
```

```
121           return self.merge_dicts(prot, ret_dict,
                  options)
122         else: # specidic calander date:
123           match = re.search(r'^Date\s+(?P<start_date>.+)
                  \s+block\s+(?P<from>\S+)\s+to\s+(?P<to>\S+)
                  \s+', rule)
124           if match:
125             options = self.interpret_options(rule)
126             ret_dict = dict(list(ret_dict.items()) +
                    list(match.groupdict().items()))
127             return self.merge_dicts(prot, ret_dict,
                    options)
128           else:
129             match = re.search(r'^Date\s+(?P<start_date
                    >.+)\s+block\s+(?P<from>\S+)', rule)
130             if match:
131               options = self.interpret_options(rule)
132               ret_dict = dict(list(ret_dict.items()) +
                      list(match.groupdict().items()))
133               return self.merge_dicts(prot, ret_dict,
                      options)
134             else:
135               return None
136
137     '''
138   Extract all the data from "Vlan" rule, return a dict
      of that data
139   '''
140     def interpret_vlan_rule(self,prot,rule):
141       ret_dict = {}
142       ret_dict["rule_type"] = "VLAN"
143       ret_dict["rule_string"] = rule
144       ret_dict["hosts"] = rule # just simple search
            later, easier the to split up the line atm
145       match = re.search(r'^Vlan\s+(?P<vlan_id>.+)\s+has'
            ,rule)
146       if match:
147         ret_dict = dict(list(ret_dict.items()) + list(
              match.groupdict().items()))
148         return self.merge_dicts(prot, ret_dict, {})
149       else:
150         return None
151
152     '''
153   Decide what kind of rule (Date, Time or Vlan) and
      send to appropriate subfuntion
154   '''
155     def interpret_primitive(self, prot, rule):
```

```
156       if "Time" in rule:
157         return self.interpret_time_rule(prot,rule)
158       elif "Date" in rule:
159         return self.interpret_date_rule(prot,rule)
160       elif "Vlan" in rule:
161         return self.interpret_vlan_rule(prot,rule)
162       elif "Data" in rule:
163         return self.interpret_data_rule(prot,rule)
164       else:
165         pass
166 
167     '''
168   This rule will return all the date from a rule by
      using the subfunctions above.
169   Its main task is to decide what protocol the rule is
      talking about, if non
170   is specified just assume TCP, UDP and ICMP is what
      the user wants.
171   '''
172     def interpret_block_and_options(self, rule):
173       prot = {}
174       if "prot TCP" in rule:
175         prot["prot"] = "TCP"
176         return self.interpret_primitive(prot, rule)
177       elif "prot UDP" in rule:
178         prot["prot"] = "UDP"
179         return self.interpret_primitive(prot, rule)
180       elif "prot ARP" in rule:
181         prot["prot"] = "ACK"
182         return self.interpret_primitive(prot, rule)
183       elif "prot ICMP" in rule:
184         prot["prot"] = "ICMP"
185         return self.interpret_primitive(prot, rule)
186       elif "Vlan" in rule:
187         prot["prot"] = "VLAN"
188         return self.interpret_primitive(prot, rule)
189       elif "Data" in rule:
190         prot["prot"] = "Data"
191         return self.interpret_primitive(prot, rule)
192       else:
193         prot["prot"] = "TCP/UDP/ICMP"
194         return self.interpret_primitive(prot, rule)
195 
196     '''
197   Reads all the policy files, extracts the policy
      varaibles
198   '''
199     def read_policy_folder(self, policy_path):
```

```
200       for file in os.listdir(policy_path):
201         if file.endswith(".pol"):
202           with open(os.path.join(policy_path,file)) as f
                :
203             lines = f.readlines()
204             for line in lines:
205               if re.match(r'^.+\s+=\s+.+',line): #
                    interpret hosts
206                 match =  re.match(r'^(.+)\s+=\s+(.+)',
                      line)
207                 self.policy_hosts_ip[match.group(2)] =
                      match.group(1)
208                 self.policy_hosts_name[match.group(1)] =
                        match.group(2)
209               elif re.match(r'^Date',line) or re.match(r
                    'Time',line)  or re.match(r'^Vlan',line
                    ) or re.match(r'^Data',line):
210                 self.policy_rules.append(line)
211                 self.policy_rules_values.append(self.
                      interpret_block_and_options(line))
212
213     '''
214   Rules can contain a "*" which specifies "any".
      Example srcip * would mean match rule with any
      source ip
215   '''
216     def check_two_values(self, rule_val, packet_val):
217       if rule_val == "*" or rule_val == None:
218         return True
219       else:
220         return rule_val == packet_val
221
222     '''
223   Compares rule src and dst against packet src and dst
224   Returns a boolean value
225   '''
226     def check_rule_and_packet(self, r_src, p_src, r_dst,
          p_dst):
227       ret1 =  self.check_two_values(r_src, p_src)
228       ret2 =  self.check_two_values(r_dst, p_dst)
229       if ret1 and ret2:
230         return ret1 and ret2
231       else:
232         ret1 =  self.check_two_values(r_dst, p_src)
233         ret2 =  self.check_two_values(r_src, p_dst)
234       return ret1 and ret2
235
```

```
236   def check_ports(self, r_sport, p_sport, r_dport,
          p_dport):
237     ret1 = self.check_two_values(r_sport, p_sport)
238     ret2 = self.check_two_values(r_dport, p_dport)
239     return ret1 and ret2
240
241   '''
242 ␣␣Finally␣checks␣if␣packet␣violates␣a␣rule␣by␣looking␣
      at␣the␣primitive␣(time,date,vlan)
243 ␣␣Returns␣True␣if␣there␣is␣a␣violation,␣false␣if␣not
244 ␣␣'''
245   def check_time_or_date(self, rule, packet,
          packet_string):
246     match = re.search(r'^(?P<date>\S+)\s+(?P<time>\S+)
          \s+', packet_string) # extract date and time
          from packet timestamp
247     if rule["rule_type"] == "Time":
248       if match:
249         mdict = match.groupdict()
250         packet_time = time.strptime(mdict.get("time"),
            "%H:%M:%S") # test this
251         rule_start_time = time.strptime(rule.get("
            start_time"),"%H:%M") # test this
252         rule_end_time = time.strptime(rule.get("
            end_time"),"%H:%M") # test this
253
254         if packet_time >= rule_start_time and
            packet_time <= rule_end_time:
255           return True
256         else:
257           return False
258     elif rule["rule_type"] == "Date":
259       if match:
260         mdict = match.groupdict()
261         packet_date = time.strptime(mdict.get("date"),
            "%a") # test this
262         rule_start_date = time.strptime(rule.get("
            start_date"),"%a") # test this
263         rule_end_date = time.strptime(rule.get("
            end_date"),"%a") # test this
264
265         if packet_date  >= rule_start_date and
            packet_date <= rule_end_date:
266           return True
267         else:
268           return False
269     else:
270       print "vlan"
```

```
271
272     '''
273   Main method for checking icmp packets, extract rule
        src and dst, packet src and dst, check if it there
        is match
274   forward matching rule and packet to next method so
        to check if values match on other parameters.
275   Lastly print out violation if any and return a
        boolean value. True if a violation, false if not.
276   '''
277     def check_icmp(self, packet_dict, packet_string):
278       for rule_dict in self.policy_rules_values:
279         if "ICMP" in rule_dict["prot"]: # rule may miss
              these keys, therefor use get -> will return
              None if key is non existent
280           r_src = None # at some point rules contains
                IPs and not host names
281           r_dst = None
282           match = re.match(r'\d{1,3}\.\d{1,3}\.\d
                {1,3}\.\d{1,3}', rule_dict.get("from"))
283           if match:
284             r_src = rule_dict.get("from")
285           else:
286             r_src = self.policy_hosts_name.get(rule_dict
                  .get("from"))
287           if rule_dict.get("to"):
288             match = re.match(r'\d{1,3}\.\d{1,3}\.\d
                    {1,3}\.\d{1,3}', rule_dict.get("to"))
289             if match:
290               r_dst = rule_dict.get("to")
291             else:
292               r_dst = self.policy_hosts_name.get(
                      rule_dict.get("to"))
293
294           if self.check_rule_and_packet(r_src,
295                                         packet_dict["
                                              ipsrc"],
                                              r_dst,
                                              packet_dict["
                                              ipdst"]):
296
297             ret = self.check_time_or_date(rule_dict,
                    packet_dict, packet_string)
298
299             if ret:
300               return rule_dict["rule_string"]
301
302     def check_udp(self, packet_dict, packet_string):
```

```
303     for rule_dict in self.policy_rules_values:
304       if "UDP" in rule_dict["prot"]: # rule may miss
             these keys, therefor use get -> will return
             None if key is non existent
305         r_src = None
306         r_dst = None
307         match = re.match(r'\d{1,3}\.\d{1,3}\.\d
               {1,3}\.\d{1,3}', rule_dict.get("from"))
308         if match:
309           r_src = rule_dict.get("from")
310         else:
311           r_src = self.policy_hosts_name.get(rule_dict
                .get("from"))
312         if rule_dict.get("to"):
313           match = re.match(r'\d{1,3}\.\d{1,3}\.\d
                {1,3}\.\d{1,3}', rule_dict.get("to"))
314           if match:
315             r_dst = rule_dict.get("to")
316           else:
317             r_dst = self.policy_hosts_name.get(
                   rule_dict.get("to"))
318
319         if self.check_rule_and_packet(r_src,
320                                       packet_dict["
                                          ipsrc"],
                                          r_dst,
                                          packet_dict["
                                          ipdst"]):
321
322           if self.check_ports(rule_dict.get("sport"),
323                               packet_dict["src_port"],
                                    rule_dict.get("dport
                                   "), packet_dict["
                                   dst_port"]):
324
325             ret = self.check_time_or_date(rule_dict,
                   packet_dict, packet_string)
326
327             if ret:
328               return rule_dict["rule_string"]
329
330   def check_tcp(self, packet_dict, packet_string):
331     for rule_dict in self.policy_rules_values:
332       if "TCP" in rule_dict["prot"]: # rule may miss
             these keys, therefor use get -> will return
             None if key is non existent
333         r_src = None
334         r_dst = None
```

```
335         match = re.match(r'\d{1,3}\.\d{1,3}\.\d
                {1,3}\.\d{1,3}', rule_dict.get("from"))
336         if match:
337           r_src = rule_dict.get("from")
338         else:
339           r_src = self.policy_hosts_name.get(rule_dict
                .get("from"))
340         if rule_dict.get("to"):
341           match = re.match(r'\d{1,3}\.\d{1,3}\.\d
                {1,3}\.\d{1,3}', rule_dict.get("to"))
342           if match:
343             r_dst = rule_dict.get("to")
344           else:
345             r_dst = self.policy_hosts_name.get(
                  rule_dict.get("to"))
346
347         if self.check_rule_and_packet(r_src,
348                                       packet_dict["
                                          ipsrc"],
                                          r_dst,
                                          packet_dict["
                                          ipdst"]):
349
350           if self.check_ports(rule_dict.get("sport"),
351                               packet_dict["src_port"],
                                    rule_dict.get("dport
                                   "), packet_dict["
                                   dst_port"]):
352
353             ret = self.check_time_or_date(rule_dict,
                  packet_dict, packet_string)
354
355             if ret:
356               return rule_dict["rule_string"]
357
358
359   def check_for_violation(self, packet_dict):
360     for vpacket in self.v_packets: # for violating
          packet in violation list
361       if vpacket["prot"] == packet_dict["prot"]:
362         if vpacket["devID"] == packet_dict["devID"]: #
              packet reply on same device
363           if vpacket["ipsrc"] == packet_dict["ipdst"]
                and vpacket["ipdst"] == packet_dict["
                ipsrc"]: # is it a reply
364             if vpacket["prot"] == "ICMP":
365               if "8" in vpacket["ping_type"] and "0"
                    in packet_dict["ping_type"]: # ping
```

```
                type is a reply
366             print "## Policy Violation Found:
                    PACKET RETURN ##"
367             print "> Packet:", vpacket["
                    packet_string"]
368             print "> Packet:", packet_dict["
                    packet_string"]
369             print "  violates:  "
370             print "> Rule:", vpacket["rule_string"
                    ]
371             if vpacket in self.v_packets:
372               self.v_packets.remove(vpacket)
373             if packet_dict in self.v_packets:
374               self.v_packets.remove(packet_dict)
375             return True
376           else:
377             if vpacket["dst_port"] == packet_dict["
                  src_port"] and vpacket["src_port"] ==
                   packet_dict["dst_port"]:
378             print "## Policy Violation Found:
                    PACKET RETURN ##"
379             print "> Packet:", vpacket["
                    packet_string"]
380             print "> Packet:", packet_dict["
                    packet_string"]
381             print "  violates:  "
382             print "> Rule:", vpacket["rule_string"
                    ]
383             if vpacket in self.v_packets:
384               self.v_packets.remove(vpacket)
385             if packet_dict in self.v_packets:
386               self.v_packets.remove(packet_dict)
387             return True
388         else: # packet is on another device
389           if vpacket["ipsrc"] == packet_dict["ipsrc"]
                and vpacket["ipdst"] == packet_dict["
                ipdst"]: # packet has been forwarded
390             if vpacket["prot"] == "ICMP":
391               if vpacket["ping_type"] == packet_dict["
                    ping_type"]: # ping type is a reply
                    of another
392             print "## Policy Violation Found:
                    PACKET FORWARDING ##"
393             print "> Packet:", vpacket["
                    packet_string"]
394             print "> Packet:", packet_dict["
                    packet_string"]
395             print "  violates:  "
```

```
396                 print ">␣Rule:", vpacket["rule_string"
                        ]
397                 if vpacket in self.v_packets:
398                   self.v_packets.remove(vpacket)
399                 if packet_dict in self.v_packets:
400                   self.v_packets.remove(packet_dict)
401                 return True
402             else:
403               if vpacket["dst_port"] == packet_dict["
                      dst_port"] and vpacket["src_port"] ==
                       packet_dict["src_port"]:
404                 print "##␣Policy␣Violation␣Found:␣
                        PACKET␣FORWARDING␣##"
405                 print ">␣Packet:", vpacket["
                        packet_string"]
406                 print ">␣Packet:", packet_dict["
                        packet_string"]
407                 print "␣␣violates:␣␣"
408                 print ">␣Rule:", vpacket["rule_string"
                        ]
409                 if vpacket in self.v_packets:
410                   self.v_packets.remove(vpacket)
411                 if packet_dict in self.v_packets:
412                   self.v_packets.remove(packet_dict)
413                 return True
414     return False
415
416   '''
417 ␣␣Recieves␣a␣string␣from␣the␣sdn_dump,␣check␣against␣
      policy␣and␣return␣boolean␣value
418 ␣␣True,␣if␣approved
419 ␣␣False,␣if␣there␣is␣a␣policy␣violation
420 ␣␣'''
421   def check_if_legal(self,dump_string):
422     #print dump_string # should check this string for
          error flags
423     if "ICMP" in dump_string:
424       match = re.match(r'^.+\s+DevID:\s+(?P<devID>.+)\
          s+(?P<prot>.+):\s+pkt\s+(?P<pktsrc>.+)\s+>\s
          +(?P<pktdst>.+),\s+ip\s+(?P<ipsrc>.+)\s+>\s
          +(?P<ipdst>.+):\s+(?P<ping_type>.+)',
          dump_string)
425       match_dict =  match.groupdict()
426       #print match_dict
427       #print match_dict
428       violation = self.check_icmp(match_dict,
          dump_string)
429       if violation:
```

```
430           string_dict = {}
431           string_dict["packet_string"] = dump_string
432           rule_dict = {}
433           rule_dict["rule_string"] = violation
434           res_dict = self.merge_dicts(match_dict,
                 string_dict, rule_dict)
435           self.v_packets.append(res_dict)
436           if self.check_for_violation(res_dict):
437             return True
438         return False
439         # now match them somehow
440       elif "UDP" in dump_string:
441         match = re.match(r'^.+\s+DevID:\s+(?P<devID>.+)\
              s+(?P<prot>.+):\s+pkt\s+(?P<pktsrc>.+)\s+>\s
              +(?P<pktdst>.+),\s+ip\s+(?P<ipsrc>.+)\s+>\s
              +(?P<ipdst>.+):\s+srcp\s+(?P<src_port>.+)\s+
              dstp\s+(?P<dst_port>.+)',dump_string)
442         match_dict =  match.groupdict()
443         #print match_dict
444         violation = self.check_udp(match_dict,
              dump_string)
445         if violation:
446           string_dict = {}
447           string_dict["packet_string"] = dump_string
448           rule_dict = {}
449           rule_dict["rule_string"] = violation
450           res_dict = self.merge_dicts(match_dict,
                 string_dict, rule_dict)
451           self.v_packets.append(res_dict)
452           if self.check_for_violation(res_dict):
453             return True
454         return False
455       elif "ARP" in dump_string: # ask if this will be
            needed, is nice to have for now
456         match = re.match(r'^.+ARP:\s+pkt\s+(?P<pktsrc
              >.+)\s+>\s+(?P<pktdst>.+),\s+hw\s+(?P<hwsrc
              >.+)\s+>\s+(?P<hwdst>.+):\s+(?P<arp_type>.+)'
              ,dump_string)
457         match_dict =  match.groupdict()
458         #print match_dict
459       elif "TCP" in dump_string:
460         match = re.match(r'^.+\s+DevID:\s+(?P<devID>.+)\
              s+(?P<prot>.+):\s+pkt\s+(?P<pktsrc>.+)\s+>\s
              +(?P<pktdst>.+),\s+ip\s+(?P<ipsrc>.+)\s+>\s
              +(?P<ipdst>.+):\s+srcp\s+(?P<src_port>.+)\s+
              dstp\s+(?P<dst_port>.+),\s+seq␣(?P<seq>.+),\s
              +ack\s+(?P<ack>.+)\s+flags\s+(?P<flags>.+)',
              dump_string)
```

```
461         match_dict =  match.groupdict()
462         #print match_dict
463         violation = self.check_tcp(match_dict,
                dump_string)
464         if violation:
465           string_dict = {}
466           string_dict["packet_string"] = dump_string
467           rule_dict = {}
468           rule_dict["rule_string"] = violation
469           res_dict = self.merge_dicts(match_dict,
                string_dict, rule_dict)
470           self.v_packets.append(res_dict)
471           if self.check_for_violation(res_dict):
472             return True
473         return False
474
475     def check_if_ports_legal(self, bsent, brecv, psent,
            precv):
476       rule_ip = None
477       for rule in self.policy_rules_values:
478         if "Data" in rule["rule_type"]:
479           if "to" in rule: # what the IP  can recv
480             match = re.match(r'\d{1,3}\.\d{1,3}\.\d
                  {1,3}\.\d{1,3}', rule.get("to"))
481             if match:
482               rule_ip = rule.get("to")
483             else:
484               rule_ip = self.policy_hosts_name.get(rule.
                    get("to"))
485             # We now know the rule IP, lets match it
                  againt actual port stats
486             for recv_stat in brecv:
487               if recv_stat == rule_ip:
488                 rule_bytes = self.
                      convert_notation_to_bytes(rule.get("
                      lim"), rule.get("notation"))
489                 if int(rule_bytes) < int(brecv[recv_stat
                      ]):
490                   print "##␣Policy␣Violation␣Found:␣
                        RECEIVED␣TO␣MUCH␣DATA␣##"
491                   print ">␣IP:", rule_ip, "address␣
                        received:", int(brecv[recv_stat])
                        , "bytes"
492                   print ">␣IP:", rule_ip, "can␣only␣
                        receive:", int(rule_bytes), "bytes
                        "
493                   print "␣␣violates:␣␣␣"
```

```
494                     print ">␣Rule:", rule.get("
                            rule_string")
495                     return True
496             elif "from" in rule: # what the IP can send
497               match = re.match(r'\d{1,3}\.\d{1,3}\.\d
                      {1,3}\.\d{1,3}', rule.get("from"))
498               if match:
499                 rule_ip = rule.get("from")
500               else:
501                 rule_ip = self.policy_hosts_name.get(rule.
                        get("from"))
502               # We now know the rule IP, lets match it
                      againsnt actual port stats
503               for recv_stat in brecv:
504                 if recv_stat == rule_ip:
505                   rule_bytes = self.
                          convert_notation_to_bytes(rule.get("
                          lim"), rule.get("notation"))
506                   if int(rule_bytes) < int(brecv[recv_stat
                          ]):
507                     print "##␣Policy␣Violation␣Found:␣
                            SENDING␣TO␣MUCH␣DATA␣##"
508                     print ">␣IP:", rule_ip, "address␣sent
                            :", int(brecv[recv_stat]), "bytes
                            "
509                     print ">␣IP:", rule_ip, "can␣only␣
                            send:", int(rule_bytes), "bytes"
510                     print "␣␣violates:␣␣␣"
511                     print ">␣Rule:", rule.get("
                            rule_string")
512                     return True
513             else:
514               pass
515       return False
516
517     def __init__(self,switch, mode):
518       self.policy_hosts_ip = {}
519       self.policy_hosts_name = {}
520       self.policy_rules = []
521       self.policy_rules_values = []
522       self.v_packets = []
523       self.read_policy_folder(os.path.dirname(os.path.
            realpath(__file__)) + "/policies")
524       if "3" in mode:
525         automator = auto.Automator(self.policy_rules,
              self.policy_rules_values,
526                                    self.policy_hosts_ip,
                                         self.
```

```
                                        policy_hosts_name,
                                        switch)
```

C.3 Automator.py

```
1  from pox.core import core
2  import pox
3
4  from pox.lib.util import dpid_to_str
5  from pox.lib.packet.ethernet import ethernet
6  import pox.openflow.libopenflow_01 as of
7  import pox.lib.packet as pkt
8  from pox.lib.addresses import EthAddr,IPAddr
9
10 import re
11 import time
12
13
14 class Automator(object):
15
16     def _handle_ConnectionUp(self,event):
17         self.switch_count += 1
18         if self.switch_count == self.switch_limit:
19             self.type_decider(event)
20
21     def __init__(self, rules, rules_values, host_ips,
          host_names, switch):
22         self.rules = rules
23         self.rules_values = rules_values
24         self.host_ips = host_ips
25         self.host_names = host_names
26         self.switch_count = 0
27         self.switch_limit = int(switch)
28         core.openflow.addListenerByName("ConnectionUp"
               , self._handle_ConnectionUp)
29         #core.openflow.addListenerByName("
               FlowStatsReceived", self.
               _handle_switch_flow_stats)
30         #core.openflow.addListenerByName("
               PortStatsReceived", self.
               _handle_switch_stats)
31
32     def hostname_to_ip_for_from(self,rule):
33         match = re.match(r'\d{1,3}\.\d{1,3}\.\d
               {1,3}\.\d{1,3}', rule.get("from"))
34         r_src = None
35         if match:
36           r_src = rule.get("from")
```

```
37          else:
38            r_src = self.host_names.get(rule.get("from")
                )
39          return r_src
40
41      def hostname_to_ip_for_to(self,rule):
42          if rule.get("to"): # if the rule has specified
                a destination
43              match = re.match(r'\d{1,3}\.\d{1,3}\.\d
                    {1,3}\.\d{1,3}', rule.get("to"))
44              r_dst = None
45              if match:
46                  r_dst = rule.get("to")
47              else:
48                  r_dst = self.host_names.get(rule.get("
                        to"))
49              return r_dst
50          else:
51              if self.hostname_to_ip_for_from(rule) == "
                    10.0.1.100":
52                  return "10.0.1.101"
53              else:
54                  return "10.0.1.100"
55
56      def get_src_port(self, rule):
57          if rule.get("sport"):
58              return rule.get("sport")
59          else:
60              return "48966" # found by doing iperf on
                    mininet
61
62      def get_dst_port(self, rule):
63          if rule.get("dport"):
64              return rule.get("dport")
65          else:
66              return "5001" # found by doing iperf on
                    mininet
67
68      def type_decider(self,event):
69          for rule in self.rules_values:
70              if "ICMP" in rule["prot"]:
71                  #print "icmp"
72                  src =  self.hostname_to_ip_for_from(
                        rule)
73                  dst =  self.hostname_to_ip_for_to(rule
                        )
74                  ping = self.create_ping(src, dst)
75                  self.send_packets(event, ping)
```

```
76                  #ping = self.create_ping(dst, src)
77                  #self.send_packets(event, ping)
78              elif "UDP" in rule["prot"]:
79                  #print "udp"
80                  src =  self.hostname_to_ip_for_from(
                        rule)
81                  dst =  self.hostname_to_ip_for_to(rule
                        )
82                  sport = self.get_src_port(rule)
83                  dport = self.get_dst_port(rule)
84                  udp = self.create_udp(src, dst, sport,
                         dport)
85                  self.send_packets(event, udp)
86              elif "TCP" in rule["prot"]:
87                  #print "tcp"
88                  src =  self.hostname_to_ip_for_from(
                        rule)
89                  dst =  self.hostname_to_ip_for_to(rule
                        )
90                  sport = self.get_src_port(rule)
91                  dport = self.get_dst_port(rule)
92                  tcp = self.create_tcp(src, dst, sport,
                         dport)
93                  self.send_packets(event, tcp)
94              else:
95                  pass
96
97      def create_udp(self, src, dst, sport, dport):
98          # Create UDP packet:
99          udp = pkt.udp()
100         udp.srcport = int(sport)
101         udp.dstport = int(dport)
102         #print "this is the udp", udp
103         # Create the IP:
104         ip = pkt.ipv4()
105         ip.protocol = ip.UDP_PROTOCOL
106         ip.srcip = IPAddr(src)
107         ip.dstip = IPAddr(dst)
108         ip.payload = udp
109         #print "THis is the ip", ip
110         return ip
111
112     def create_tcp(self, src, dst, sport, dport):
113         # Create TCP:
114         tcp = pkt.tcp()
115         tcp.srcport = int(sport)
116         tcp.dstport = int(dport)
117         tcp._setflag(tcp.SYN_flag,1)
```

```
118         tcp.seq = 0
119         tcp.ack = 0
120         tcp.win = 1
121         tcp.off = 5
122         #print tcp
123 
124         # Create the IP:
125         ip = pkt.ipv4()
126         ip.protocol = ip.TCP_PROTOCOL
127         ip.srcip = IPAddr(src)
128         ip.dstip = IPAddr(dst)
129         ip.payload = tcp
130         #print ip
131         return ip
132 
133     def create_ping(self, src, dst):
134         # Make a ping request:
135         icmp = pkt.icmp()
136         icmp.type = pkt.TYPE_ECHO_REQUEST
137         echo = pkt.ICMP.echo(payload = "0123456789")
138         icmp.payload = echo
139         #print "THis is the ping:", icmp
140 
141         #Create IP packet
142         ip = pkt.ipv4()
143         ip.protocol = ip.ICMP_PROTOCOL
144         ip.srcip = IPAddr(src)
145         ip.dstip = IPAddr(dst)
146         ip.payload = icmp
147         #print "THis is the ip", ip
148         return ip
149 
150     def send_packets(self, event, ip_packet):
151         #Create Ethernet Payload
152         eth = ethernet()
153         eth.src = EthAddr("ff:ff:ff:ff:ff:ff")
154         eth.dst = EthAddr("ff:ff:ff:ff:ff:ff")
155         eth.type = eth.IP_TYPE
156         eth.payload = ip_packet
157         #print "This is the ethenret", eth
158 
159         msg = of.ofp_packet_out()
160         #msg.actions.append(of.ofp_action_output(port
                = 1))
161         msg.data =  eth.pack()
162         msg.in_port = of.OFPP_NONE
163         #for i in range(5): # send packet untill 5
                port ranges
```

```
164             #msg.actions.append(of.ofp_action_output(
                    port = i + 1))
165         msg.actions.append(of.ofp_action_output(port =
                of.OFPP_CONTROLLER))
166         event.connection.send(msg)
167             #core.openflow.getConnection(event.dpid).
                    send(msg)
```

C.4 __init__.py

```
1 # Module file
```

C.5 Example policy

```
1  # Hosts
2  host1 = 10.0.0.1
3  host2 = 10.0.0.2
4
5  # Time blocks
6  Time 10:00 to 23:00 block host1 to host2
7  Time 10:00 to 23:00 block host1 prot TCP
8
9  # Date blocks
10 Date MON to SUN block host1 to host2
11 Date MON to FRI block 10.0.1.100
12 Date MON to FRI block host2
13 Date MON to FRI block host1 to 10.0.0.3
14
15
16 # Data rule
17 Data 1 KB to host1
18 Data 1000 b from host2
19 Data 10 KB to 10.0.0.100
20
21 # Vlans
22 Vlan 10 has h1, h2
```